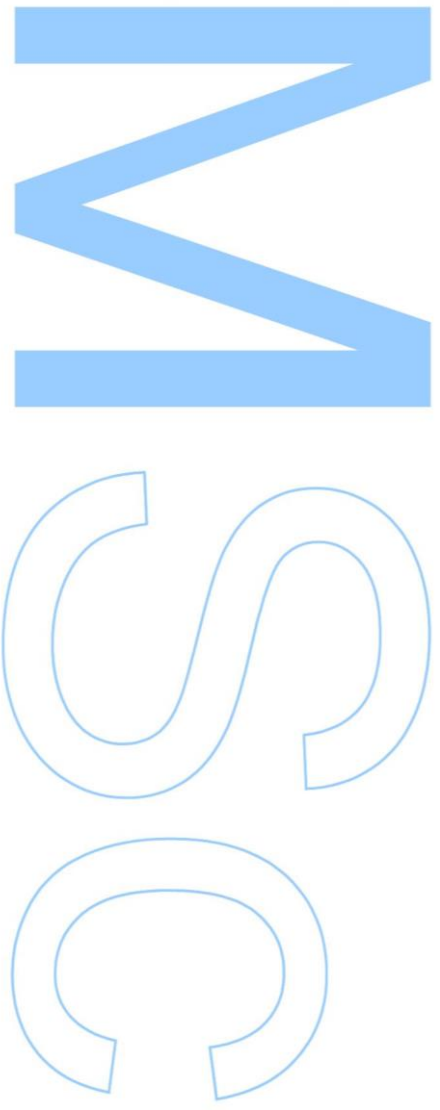
Algorithms for Chromatic Art Gallery Problems with Vertex α-Guards

Catarina Lobo do Souto Ferreira
Master's degree in Computer Science
Computer Science Department
2015/2016

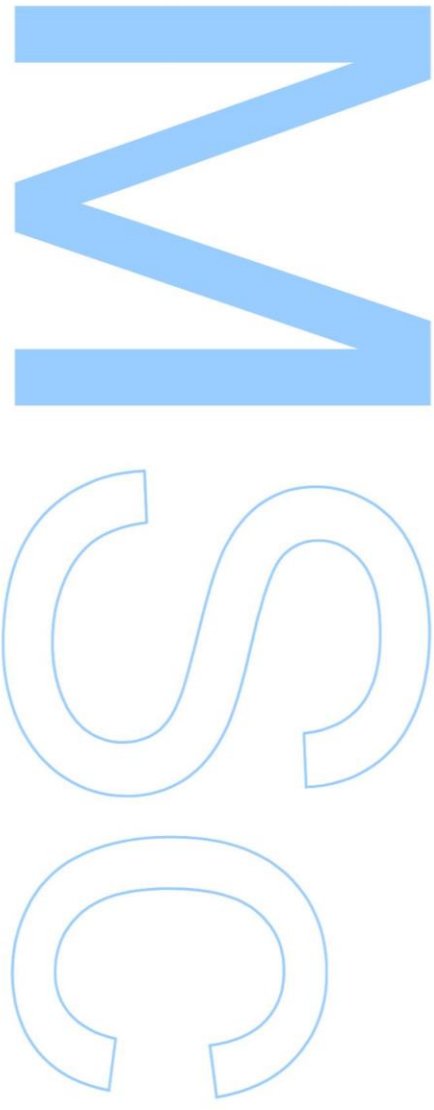
Supervisor
Ana Paula Tomás, Associate Professor, Faculty of Sciences of University of Porto



All corrections determined by the jury, and only those, were incorporated.

The President of the Jury,

Porto, _____/_____/_____



Catarina Lobo do Souto Ferreira

Algorithms for Chromatic Art Gallery Problems with Vertex α-Guards

Dissertation submitted to the Faculty of Sciences
University of Porto as part of the requirements for the degree of
Master in Computer Science

Supervisor: Prof. Ana Paula Tomás

Department of Computer Science
Faculty of Sciences University of Porto
September 2016

To my grandfather Fano

Acknowledgements

I would like to first acknowledge Professor Ana Paula Tomás, who brightly oriented me along the whole thesis. The many precious advices and the persistence to look for the perfection will be kept in mind for my future. I will not forget the long (very long) meetings that we had, working and discussing every emerged interesting issues. I want to enhance the confidence that Professor showed on me, on my work and effort.

I want to thank a person very special, Miguel Silva, who was always present and supported me in uncountable problems that appeared during the years. I also want to thank the insistence to wait for me until the end of my long (very long) meetings.

I want to show my gratitude to my family, mainly to my grandfather, who ever looked for my future and never tired to give me wiser advices and, to my grandmother, I am really grateful for all the warm receptions whenever I visited them. To my uncle Pedro Souto, I am thankful for all the help, the important suggestions and the active presence on my academic life. To my mother, I am very grateful for the education, the dedication to not let me feel unhappy and all the kindness. To my siblings, thank you for every single funny moment!

To my friends and colleagues, I want to thank for all the good moments that we shared.

I would like to acknowledge Professors Fernando Silva, Mário Florido, Sabine Broda and Sandra Alves for considering and supporting my work and my pleasure to learn during the course.

Catarina Lobo

Porto, 2016

Abstract

The Chromatic Art Gallery Problem (CAGP) is a recent problem in the area of computational geometry with potential practical applications. It can be seen as a variant of the classical Art Gallery Problem (AGP). Both problems involve stationing guards in a (polygonal) environment P so that each point of P is seen by at least one guard. In CAGP each guard is assigned a color and no two guards with the same color have overlapping visibility regions. While AGP asks for the minimal number of guards, CAGP asks for the minimum number of colors. Many variations of AGP have been studied over the years to deal with various types of constraints on guards and different notions of visibility.

In this thesis, we focused on the Chromatic Art Gallery Problem with α-guards (α-CAGP), a variant of CAGP with guards whose range of vision is limited to some angle α, for edge-aligned vertex guards. We restrict to orthogonal polygons and $\alpha = \frac{\pi}{2}, \pi, 2\pi$. This problem has been studied in the area of AGP, but it is novel in CAGP. We show combinatorial tight bounds for the chromatic α-guard number for the Steiner path orthogonal polygons. In addition, we develop a prototype solver for α-CAGP, for generic instances of orthogonal polygons. The solver performs a discretization of the problem, using geometric software, finds a 0-1 Integer Linear Programming (ILP) model for the discretized version, and makes use of optimization software for finding an optimal solution.

In the first part, we address the generation of random orthogonal polygons, based on the Inflate-Paste technique. We developed a C++ library for generic and specific families of orthogonal polygons, that is compatible with the CGAL. Finally, we integrated the α-CAGP solver with the generator to carry out an empirical study, with the goal of

proving or disproving some conjectures about the chromatic α-guard number of the path orthogonal polygons. We present some preliminary results from this study.

Keywords: orthogonal polygons, polygon generation, surveillance and visibility, chromatic art gallery problems, α-guards, geometric algorithms, optimization

Resumo

O Problema Cromático da Galeria de Arte (CAGP) é um problema recente na área de Geometria Computacional, com aplicações práticas. Pode ser visto como uma variante do Problema da Galeria de Arte (AGP) clássico. Ambos requerem a colocação de guardas numa região (poligonal) de modo que todos os pontos possam ser observados. No CAGP, cada guarda tem uma cor e dois guardas têm cores distintas se as suas regiões de visibilidade se sobrepuserem. Enquanto que AGP procura minimizar o número de guardas, CAGP tem como objetivo minimizar o número de cores. De modo a modelar vários tipos de restrições, foram introduzidas diversas variantes de AGP ao longo dos anos.

Na tese, abordamos α-CAGP, o Problema Cromático da Galeria de Arte com guardas com visão limitada a um ângulo α. Este problema foi considerado na área de AGP, mas é novo em CAGP. Os guardas são colocados em vértices e o cone de visibilidade destes está alinhado com uma das arestas incidentes. Neste trabalho, focamos apenas em polígonos ortogonais e com $\alpha = \frac{\pi}{2}, \pi, 2\pi$.

Obtivemos limites exatos para o número cromático de α-guardas para os polígonos ortogonais *Steiner path*. Desenvolvemos também um protótipo para resolução de α-CAGP para instâncias genéricas de polígonos ortogonais, baseado na modelação de α-CAGP como um problema de Programação Linear Inteira (ILP) com variáveis 0-1. O processo de discretização é suportado por um software geométrico e a procura da solução ótima por um software de otimização.

Na primeira parte da tese, abordamos a geração de polígonos ortogonais aleatórios, usando a técnica Inflate-Paste. Desenvolvemos uma biblioteca em C++ para gerar polígonos ortogonais e subfamílias, a qual é compatível com a biblioteca CGAL. Final-

mente, integrámos o resolutor de α-CAGP com o gerador para proceder a um estudo empírico, para análise de algumas conjeturas sobre o número cromático de α-guardas em polígonos ortogonais da família *path*. Apresentamos alguns resultados preliminares desse estudo.

Palavras-Chave: polígonos ortogonais, geração de polígonos, vigilância e visibilidade, problema da galeria de arte cromático, α-guarda, algoritmos geométricos, otimização

Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

α-CAGP Chromatic Art Gallery Problem with α-guards. 2, 3, 8, 9, 53, 55, 59, 69, 70, 71, 73, 79, 83, 84

AGP Art Gallery Problem. 1, 2, 8, 10, 11, 69, 72

AVP Atomic Visibility Polygon. 73, 74, 75, 76, 77

BIP Binary Integer Programming. 69

CAGP Chromatic Art Gallery Problem. 1, 2, 8, 9, 10, 11, 12, 66, 69, 70, 72

CCW counterclockwise. 4, 16, 21, 27, 75, 76, 83

CGAL Computational Geometric Algorithms Library. 3, 69, 74, 83

CW clockwise. 75

DCEL Doubly-Connected Edge List. 75

FIP α-Floodlight Illumination Problem. 8

GLPK GNU Linear Programming Kit. 69

ILP Integer Linear Programming. 2, 69, 72

Chapter 1

Introduction

The Art Gallery Problem (AGP) is a well-studied problem in Computational Geometry, modeling the problem of guarding paintings in an art gallery. The floorplan of the gallery can be seen as a polygon with edges matching the room walls. The guards stand at points inside the polygon.

According to Ross Honsberger [19], AGP was originally posed by Victor Klee in 1973, as the problem of determining the minimum number of guards sufficient to cover the walls on a n-wall art gallery. Nowadays, AGP refers to the problem of covering not only the walls but also the interior of the gallery.

Some other real-world problems involving cost optimization can be modeled in terms of AGP, such as the minimization of the number of security cameras in a shop or of the number of lights needed to illuminate an environment.

Many variations of art gallery problems have been studied over the years to deal with various types of constraints on guards and different notions of visibility. The primitive AGP is defined assuming *point guards* with omnidirectional unlimited vision (i.e., 2π-guards). Point guards are stationary guards that can be placed at any point in the polygon, whereas *edge guards* and *vertex guards* can be placed at edges and vertices only. The term edge-guard is often used to define a mobile guard that can move along an edge. In this thesis, we consider vertex guards only. Variants of AGP with guards whose range of vision is limited to some angle α were also studied (e.g., [32, 29, 18]).

In 2010, Erickson and LaValle introduced a colored variant of AGP, called the Chromatic Art Gallery Problem (CAGP) [8], in which guards are colored and no two guards with the same color can have overlapping visibility regions. CAGP aims at the minimization of the number of colors. For the discrete case, the guards to be used to cover the polygon are selected from a given finite set.

CAGP serves as a model for a landmark-based navigation problem. In a closed environment with landmarks spread inside it, a robot navigates while seeing a particular landmark at the moment. The environment is represented by a polygon and the landmarks by guards. This landmark system has to ensure that wherever the robot is placed, it has at least one landmark to communicate with. Moreover, the robot needs to distinguish the landmarks (different colors or transmission frequencies) that it is seeing currently in order to avoid ambiguity. Thus, the landmark system has also to guarantee that at any site that the robot can stay, all the landmarks capable to communicate to it have different colors. This landmark-based navigation system becomes more simple and less costly once we minimize the number of colors for the landmarks.

The minimization of the number of colors and the minimization of the number of guards are essentially distinct problems. Indeed, there exist polygons for which no AGP-optimal guard set leads to an optimal solution of CAGP.

For wireless landmark-based navigation, landmarks with a more reduced wireless range can be less expensive than the ones with a 2π-range. Such practical issue is a motivation for this thesis. We define Chromatic Art Gallery Problem with α-guards (α-CAGP), a variant of CAGP with guards whose range of vision is limited to some angle α, for edge-aligned vertex guards. We restrict to orthogonal polygons and $\alpha = \frac{\pi}{2}, \pi, 2\pi$. This problem has been studied in the area of AGP, but it is novel in CAGP.

We study α-CAGP in Steiner path orthogonal polygons, from a combinatorial point of view, and derive combinatorial tight bounds. For 2π-CAGP, lower and upper bounds on other families of orthogonal polygons are shown in [8, 9]. In addition, we developed a prototype solver for α-CAGP, for generic instances of orthogonal polygons. The solver performs a discretization of the problem, using geometric software, finds a 0-1

Integer Linear Programming (ILP) model for the discretized version, and makes use of optimization software for finding an optimal solution.

In the first part of the thesis, we address the generation of random orthogonal polygons based on the Inflate-Paste technique, proposed in [30]. We developed and implemented a C++ library for generic and specific families of orthogonal polygons. We tried to make this package consistent with the coding conventions of the Computational Geometric Algorithms Library (CGAL)[1], since we would like to share it to this library. To carry out experimental work on α-CAGP that could support our theoretical studies, or to obtain statistical data that help to predict the number of colors required, we integrated the α-CAGP solver with the generator of polygons. We present some preliminary results from empirical tests that aimed at proving or disproving our conjectures about the chromatic α-guard number of the path orthogonal polygons.

The first part of the thesis (Chapters 2 and 3) describes in detail the implementation of the generator of orthogonal polygons and its customization for particular classes of orthogonal polygons. The second part (Chapters 4 and 5) presents our results for α-CAGP on Steiner path orthogonal polygons and the development of the prototype solver for α-CAGP.

In the next section we introduce some basic definitions used throughout the thesis, namely about orthogonal polygons and polyominoes. Then, we present α-CAGP. In the thesis, we focused on simple orthogonal polygons (that is, orthogonal polygons without holes).

1.1 Orthogonal Polygons and Polyominoes

A *polygon* P in the plane is a closed subset of $\mathbb{R}^2$ delimited by finite chains of straight line segments, that form a polygonal curve (see Figure 1.1). Non-adjacent segments do not intersect and adjacent segments intersect only in their common endpoint. Such endpoints are the *vertices* of P and the line segments are the *edges* of P. The boundary of P, i.e., the polygonal curve is denoted by ∂P. In the sequel, n stands for number of vertices of the polygon.

[1] http://www.cgal.org/

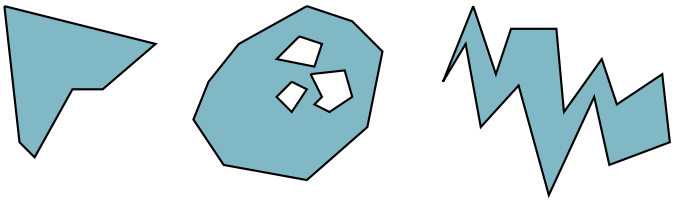


Figure 1.1: Three polygons.

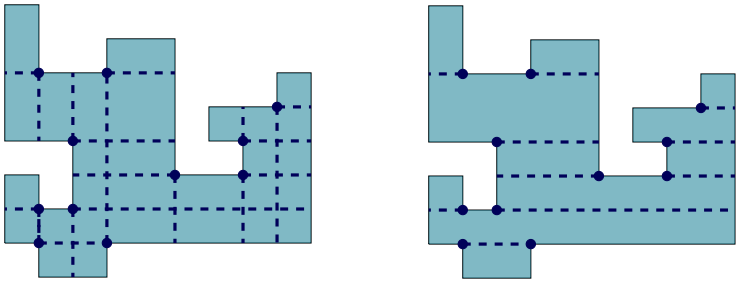
A *simple polygon* contains no holes. In Figure 1.1, the polygon in the middle contains three holes (which are simple polygons). Both the other ones are simple polygons.

A vertex is *convex* if the interior angle between its two incident edges is at most π and is *reflex* (or *concave*) otherwise. In this thesis, r represents the number of the reflex vertices, when we refer to the number of vertices of a polygon.

A polygon is *orthogonal* (or *rectilinear*) if its edges always meet orthogonally, with internal angles of $\frac{\pi}{2}$ and $\frac{3\pi}{2}$ radians (right angles). We assume that orthogonal polygons have horizontal and vertical edges, aligned with a pair of orthogonally coordinate axis. A n-vertex orthogonal polygon is also known as *n-ogon*, for short. The number of vertices of an orthogonal polygon is even. O'Rourke [24] has shown that $n = 2r + 4$, for every n-ogon with r reflex vertices.

Throughout the thesis, we use H and V as an abbreviation for *horizontal* and *vertical*. Hence, a H-edge is an horizontal edge, a V-line is a vertical line, and so on. We consider that the sequence of vertices of a polygon is given in counterclockwise (CCW) orientation. For any given vertex v of an orthogonal polygon, we denote the horizontal edge incident to v by $e_H(v)$ and the vertex that is adjacent to v in that edge by $a_H(v)$. In a similar way, $e_V(v)$ stands for the V-edge incident to v and $a_V(v)$ is the vertex that is adjacent to v in that edge. We use (p_x, p_y) to denote the coordinates of a point p.

The *rectilinear partition* $\Pi_{HV}(P)$ of an orthogonal polygon P is the one we obtain by adding all the *rectilinear* (horizontal and vertical) cuts (or chords) incident to the reflex vertices of P, that is, by extending the edges incident to a reflex vertex towards the interior of P until they reach the boundary again (see Figure 1.2(a)). Every piece in $\Pi_{HV}(P)$ is a rectangle, which we call a *r-piece*.

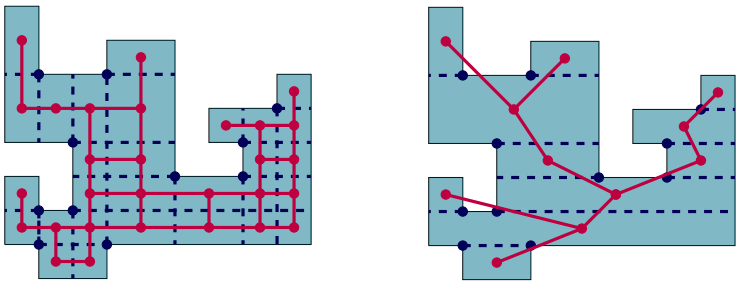


(a) (b)

Figure 1.2: Orthogonal polygon with its: (a) rectilinear partition, (b) horizontal partition.

When we add the horizontal cuts only, we get the *horizontal partition* $\Pi_H(P)$ (see Figure 1.2(b)). Similarly, we obtain the *vertical partition* $\Pi_V(P)$ if we add the vertical cuts only.

By definition, the *dual graph of a partition* Π is the graph whose vertices correspond to the pieces of Π and whose edges connect those pieces of Π that share an edge (Figure 1.3).



(a) (b)

Figure 1.3: (a) Dual graph of the rectilinear partition of an orthogonal polygon. (b) Dual graph of the horizontal partition of an orthogonal polygon.

A *polyomino* is a polyform constructed from unit squares joined edge-to-edge on a regular two-dimensional Cartesian plane. We restrict to polyominoes that can be viewed as a partition of a simple polygon (see Figure 1.4).

Therefore, we can map polyominoes to rectilinear polygons and any rectilinear polygon whose vertices lie on a *lattice* (unit square grid) can be represented by a polyomino.

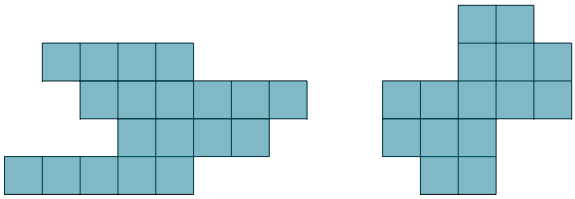


Figure 1.4: Examples of polyominoes.

A polyomino is row-convex (or horizontally convex) if the set of cells in each horizontal row is connected (i.e., consists of a single block of contiguous cells). This definition can be rephrased in terms of the concept of y-monotonicity.

A polygon P is *monotone with respect to a line* L (or simply L-*monotone*) if the intersection of P with any line L' perpendicular to L has at most one connected component, i.e., $P \cap L'$ is a segment, a point or an empty set. In Figure 1.5, we illustrate this notion, assuming that L is a vertical line.

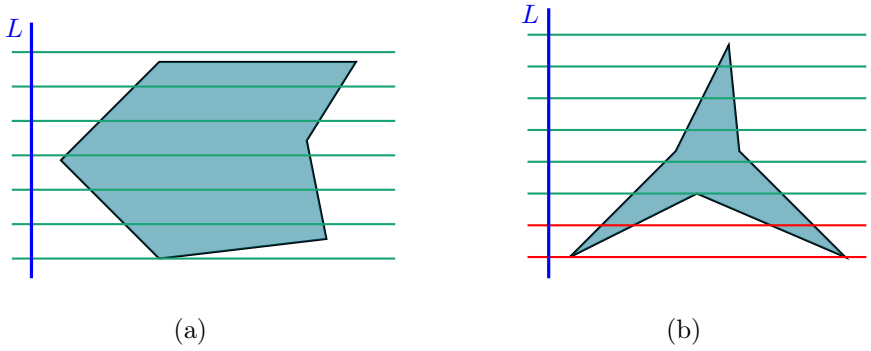
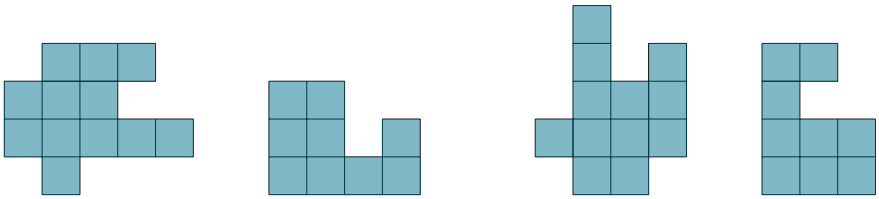


Figure 1.5: (a) A polygon that is monotone with respect to L. (b) A polygon that is not monotone w.r.t. to L because of the horizontal lines colored by •.

A polygon P is x-*monotone* when it is monotone with respect to an horizontal line. Similarly, P is y-*monotone* when it is monotone with respect to a vertical line. We can assume these lines to be the x-axis and the y-axis, which justifies the terminology used.

Hence, a polyomino is row-convex if the associated orthogonal polygon is y-monotone. The definition of *column-convex polyomino* (*vertically convex polyomino*) is similar. Each column of a column-convex polyomino consists of a single block of contiguous cells, meaning that the associated polygon is x-monotone. Figure 1.6 illustrates these notions.



(a) (b)

Figure 1.6: (a) Row-convex and not row-convex polyominoes. (b) Column-convex and not column-convex polyominoes.

A *convex* polyomino is both horizontally and vertically convex (Figure 1.7).

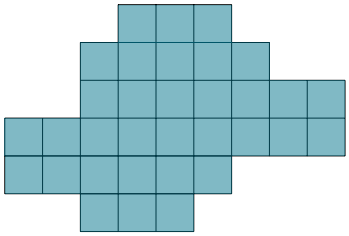


Figure 1.7: A convex polyomino.

Here, convex stands for *orthogonally convex*. Actually, no orthogonal polygon with $n \geq 6$ vertex is convex in the usual sense, which requires that any straight line segment that links two points in the polygon belongs to the polygon.

The convex polyominoes are the ones for which the perimeter of its minimum bounding rectangle equals the perimeter of the corresponding orthogonal polygon [3].

If a convex polyomino contains two vertices as opposite corners of the minimum bounding rectangle, it is called a *staircase* polyomino. Figure 1.8 exemplifies these notions.

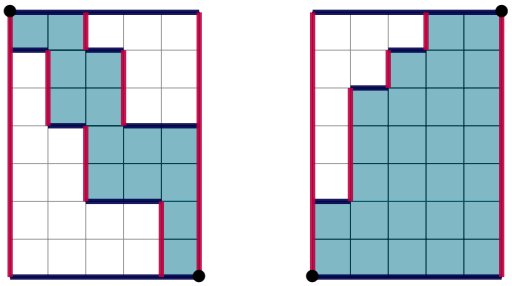
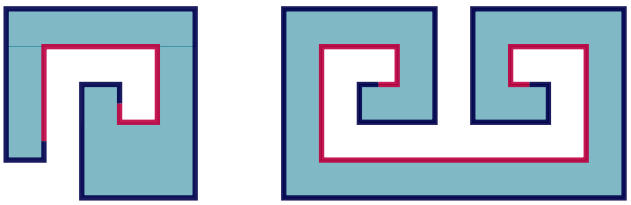


Figure 1.8: Staircase polyominoes with the same minimum bounding rectangle.

A *spiral* n-ogon is a n-vertex orthogonal polygon whose boundary can be divided into two chains, a reflex and a convex vertex chain (see Figure 1.9).



(a) (b)

Figure 1.9: (a) A spiral 12-ogon. (b) A spiral 20-ogon. Reflex chains are hued by • and convex chains by •.

1.2 The Chromatic Art Gallery Problem with α-Guards

In the general visibility model, two points p and q in a polygon P *see* each other if the line segment $\overline{pq}$ contains no points of the exterior of P. Although most of the work on AGP has focused on guards with 2π-range of vision, art gallery problems involving angle guards (also known as *floodlights*) have been addressed also. In the α-Floodlight Illumination Problem (FIP), the visibility from a point p is limited to a cone with aperture α and apex p.

In this section, we give a formal description of α-CAGP, a variant of CAGP with α-guards (we use the term α-guard instead of α-floodlight). Our definitions extend the original ones for CAGP to cope with guards with a limited range of vision.

Given a point p in polygon P and a fixed cone C_α^p with aperture α, for $\alpha \in]0, 2\pi]$, and apex p, a point q is *visible* from p if $\overline{pq} \subseteq P \cap C_\alpha^p$ (see Figure 1.10).

The *visibility region* of p with range C_α^p is given by $\mathcal{V}(p, C_\alpha^p) = \{q \mid \overline{pq} \subseteq P \cap C_\alpha^p\}$. We note that, for this notion, the visibility relation is not symmetric, i.e., $p \in \mathcal{V}(q, C_\alpha^q)$ does not imply $q \in \mathcal{V}(p, C_\alpha^p)$.

Let G be a subset of P and C_α^G be a set of cones with the same aperture α, for $\alpha \in]0, 2\pi]$, such that every $g \in G$ is associated to exactly one fixed cone $C_\alpha^g \in C_\alpha^G$. The region

visible to (or covered by) G is given by

$$\mathcal{V}(G, C_\alpha^G) = \bigcup_{g \in G, C_\alpha^g \in C_\alpha^G} \mathcal{V}(g, C_\alpha^g).$$

It consists of the set of all points that are visible to at least one point in G, for the visibility cones defined by C_α^G. When G is finite and $\mathcal{V}(G, C_\alpha^G) = P$, we say that (G, C_α^G) is a *α-guard set of P*, i.e., it *covers* P. We call $\mathcal{G}_\alpha$ a *candidate α-guards set of P* if there is at least one pair (G, C_α^G) in $\mathcal{G}_\alpha$ that is a α-guard set of P.

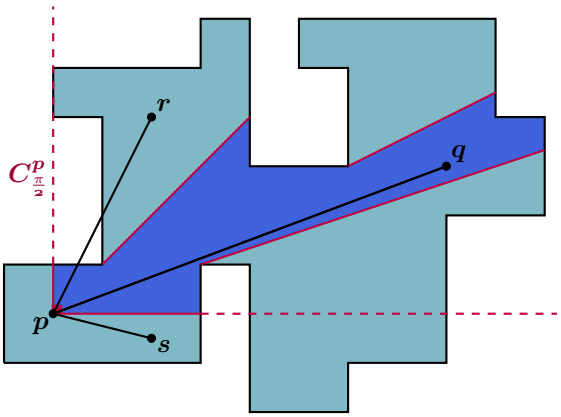


Figure 1.10: The visibility region of point p with a cone $C_{\frac{\pi}{2}}^p$ (color •). $\overline{pq} \subseteq P \cap C_{\frac{\pi}{2}}^p$, but $\overline{pr} \not\subseteq P$ and $\overline{ps} \not\subseteq C_{\frac{\pi}{2}}^p$.

Two guards (p, C_α^p) and (q, C_α^q) *conflict* if there is a point that is visible from both, i.e., if $\mathcal{V}(p, C_\alpha^p) \cap \mathcal{V}(q, C_\alpha^q) \neq \emptyset$. For (G, C_α^G) in $\mathcal{G}_\alpha$, the *conflict graph of* (G, C_α^G), denoted by $\mathcal{C}(G, C_\alpha^G)$, is the graph with vertex set G and edge set $E = \{(g_1, g_2) \mid (g_1, C_\alpha^{g_1})$ *and* $(g_2, C_\alpha^{g_2})$ $conflict\}$. The *chromatic number of* this graph, denoted by $\mathcal{X}(G, C_\alpha^G)$, is the minimum number of colors required to color the vertices of the graph in such a way that all adjacent vertices have distinct colors.

We can establish a natural relation between such a coloring and CAGP. The *chromatic α-guard number of P* is given by

$$\mathcal{X}_{\mathcal{G}_\alpha}(P) = \min_{(G, C_\alpha^G) \text{ in } \mathcal{G}_\alpha \text{ and } \mathcal{V}(G, C_\alpha^G) = P} \mathcal{X}(G, C_\alpha^G).$$

Then, α-CAGP aims at finding the chromatic α-guard number of a given polygon P and a α-guard set of P, (G, C_α^G), that $\mathcal{X}_{\mathcal{G}_\alpha}(P) = \mathcal{X}(G, C_\alpha^G)$.

Since the aperture of a visibility cone can be less than 2π, it could have uncountable many inclinations (Figure 1.11).

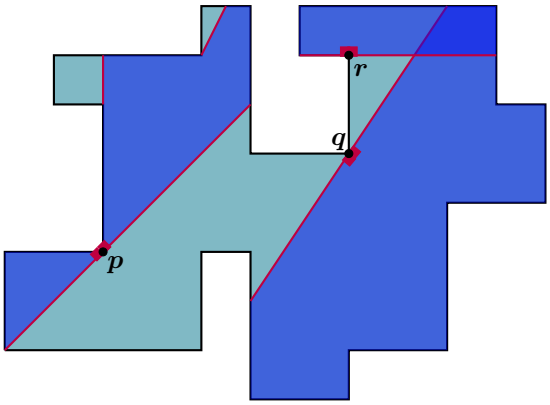


Figure 1.11: Three types of π-guards. The guard at point p is general, the one at q is inward-facing and the other at r is edge-aligned.

A *general* α-guard (g, C_α^g) is one without any restrictions on the orientation of C_α^g. Some works consider inward-facing and edge-aligned guards, for guards at vertices. For an *inward-facing* α-guard, the cone C_α^g is turned towards the interior of P, i.e., the edges of P incident to g are disjoint from the interior of C_α^g. An *edge-aligned* α-guard is inward-facing and at least one of the bounding rays of its cone C_α^g is aligned with an edge of P. We observe that a 2π-guard is always a general 2π-guard.

In this thesis, we focus on *edge-aligned vertex α-guards*, for $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$. So, G is the set of vertices of P, for all (G, C_α^G), and the cone C_α^v is edge-aligned, for all v.

1.3 State of the Art

Over the years, research on AGP and CAGP has led to combinatorial bounds, algorithms and hardness results for several variations of the problem, for either different notions of visibility or types of constraints on guards (for surveys, we refer to e.g. [25, 32, 29, 7, 14]). In this section, we give a brief overview of some of these results.

In 1975, Chvátal [5] proved that $\lfloor \frac{n}{3} \rfloor$ point 2π-guards are always sufficient and sometimes necessary to cover any n-vertex simple polygon. This result is known as the *Art Gallery Theorem*. In 1978, Fisk [15] gave a much simpler and elegant proof of this upper bound, for vertex 2π-guards, by reducing the problem to the problem of coloring the graph induced by a triangulation of the polygon. In 1983, Kahn, Klawe

and Kleitman [21] showed that at most $\lfloor \frac{n}{4} \rfloor$ vertex 2π-guards are required to cover any n-vertex simple orthogonal polygon and that this bound is tight, as well.

In 1986, Lee and Lin [22] proved some hardness results on the computational complexity of AGP variants. They proved the NP-hardness of AGP with point 2π-guards and vertex 2π-guards for simple polygons by a reduction from 3-SAT. In 1995, Schuchardt and Hecker [28] showed the NP-hardness of AGP with point 2π-guards and vertex 2π-guards, for simple orthogonal polygons by a similar reduction from 3-SAT.

Guards with an aperture less than 2π have been also exploited. For an interesting survey, although from 1995, we refer to a paper by Urrutia [32]. Estivill-Castro et al. [12] showed that there exist simple polygons that cannot be guarded by general vertex $(\pi - \epsilon)$-guards, for every $\epsilon > 0$. Estivill-Castro and Urrutia [11] proved that, for simple n-vertex orthogonal polygons coverage, $\lfloor \frac{3n-4}{8} \rfloor$ edge-aligned vertex $\frac{\pi}{2}$-guards are sufficient and sometimes necessary. Abello et al. [1] proved that $\lfloor \frac{3n+4(h-1)}{8} \rfloor$ edge-aligned vertex $\frac{\pi}{2}$-guards are sufficient and sometimes necessary to cover n-vertex orthogonal polygons with $h \geq 1$ holes. In addition, they proved that there are simple orthogonal polygons that cannot be covered by general vertex $(\frac{\pi}{2} - \epsilon)$-guards, for all $\epsilon > 0$. This result is interesting for our work because it implies that, in general, it is not possible to restrict the aperture α arbitrarily (i.e., we must fix $\alpha \geq \frac{\pi}{2}$).

In 2003, Speckmann and Tóth [29] showed another interesting bound: any simple polygon with k convex vertices can be monitored by at most $\lfloor \frac{2n-k}{3} \rfloor$ edge-aligned vertex π-guards and this number is tight. From this result, we can conclude that $\frac{n}{2} - 1$ edge-aligned vertex π-guards are always sufficient to guard a simple n-vertex orthogonal polygon. Nevertheless, this bound is not too interesting as, from the aforementioned result by Abello et al., we deduce that $\lfloor \frac{3n}{8} \rfloor - 1$ edge-aligned vertex π-guards are already sufficient for simple n-ogons.

Erickson and LaValle introduced CAGP in 2010 and obtained some combinatorial bounds and complexity results [8, 9, 10]. In particular, they showed that $\mathcal{X}_{\mathcal{G}_{2\pi}}(P) \leq 2$, for any spiral polygon P, assuming edge 2π-guards. For any staircase polygon P, they showed that $\mathcal{X}_{\mathcal{G}_{2\pi}}(P) \leq 3$ if we use vertex 2π-guards. For generic simple polygons, they constructed a family of polygons with $4k$ vertices requiring at least k colors, for point 2π-guards, for $k \geq 3$. They showed also that, for every odd integer $k \geq 3$, there is a

monotone orthogonal polygon P with $4k^2 + 10k + 10$ vertices such that $\mathcal{X}_{\mathcal{G}_{2\pi}}(P) \geq k$, using point 2π-guards. Moreover, they showed that, if $\mathcal{G}_{2\pi}$ is a finite set, then the construction of Lee and Lin [22] can be used to prove the NP-hardness of determining $\mathcal{X}_{\mathcal{G}_{2\pi}}(P)$, for a simple polygon P, using a reduction from 3-SAT.

In 2014, Zambon et al. [34] provided an Integer Programming formulation for the discrete version of CAGP and discussed some techniques to improve the performance of the solver.

In the same year, Fekete, Friedrichs and Hemmer [13] showed that it is NP-hard to decide if three or more colors are sufficient for a polygon P with holes, assuming point 2π-guards. More precisely, using a reduction from k-coloring of planar graphs, they showed that deciding whether $\mathcal{X}_{\mathcal{G}_{2\pi}}(P) \geq k$, for $k > 2$, is NP-hard. They proved also that it is already NP-hard to decide if two colors suffices for P, that is if $\mathcal{X}_{\mathcal{G}_{2\pi}}(P) = 2$, by a reduction from 3-SAT. In his PhD thesis, Erickson [7] presented an additional important complexity result for a simple polygon P, assuming point 2π-guards, namely that deciding whether $\mathcal{X}_{\mathcal{G}_{2\pi}}(P) \geq 5$ is NP-hard. For the proof, he constructed a reduction from the 4-coloring problem in circle graphs.

More recently, in 2015, Hoorfar and Mohades [20] defined a CAGP variant for guards with limited range of vision and proved various upper bounds on the number of colors for simple polygons, for a range less than or equal to π. However, they assume that the visibility cones do not include the supporting rays and they also allow to locate more than one guard at the same point. This makes the problem distinct to the one we address in the thesis.

Chapter 2

Grid n-Ogons Generation

In this chapter and the following one, we present the algorithms we developed and implemented for the generation of generic orthogonal simple polygons and of some specific families of the orthogonal polygons. All these algorithms are based on the *Inflate-Paste* technique [30]. This construction technique was proposed for the generation of *grid orthogonal polygons*, a.k.a. permutominoes (*grid ogons*, for short). A grid ogon is an orthogonal polygon without collinear edges, defined in a unit square lattice and that has exactly one edge on all the grid lines that intersect their minimum bounding square. This class is generic enough because we can use grid ogons to create generic orthogonal polygons, by spacing the grid lines arbitrarily (or sliding edges), while preserving their relative order somehow.

The structural properties of the horizontal partition of such polygons are exploited in the design of Inflate-Paste [30]. We proceed in a similar way for adapting the method to yield grid ogons of more specific subclasses, namely row-convex, convex, thin and spiral polygons.

2.1 The Inflate-Paste Algorithm for Grid Ogons

In this section, we present the definition of grid n-ogons, their relation to generic orthogonal polygons and recall the Inflate-Paste method.

An n-ogon P is in *general position* if and only if every horizontal and vertical line has at most one edge of P, i.e., if and only if P has no collinear edges. A *grid n-ogon* corresponds to a polyomino in general position defined in a $\frac{n}{2} \times \frac{n}{2}$ grid. Since an n-ogon has $\frac{n}{2}$ horizontal and $\frac{n}{2}$ vertical edges, we have the following result.

Lemma 1. *Each grid n-ogon has exactly one edge in every line of a $\frac{n}{2} \times \frac{n}{2}$ bounded grid.*

An n-ogon not in general position can be mapped to an n-ogon in general position by performing ϵ-perturbations (ϵ-deflections), for a sufficiently small constant $\epsilon > 0$. On the other hand, every n-ogon P in general position may be identified with a unique grid n-ogon, by performing a planar sweep (see Figure 2.1 for an example).

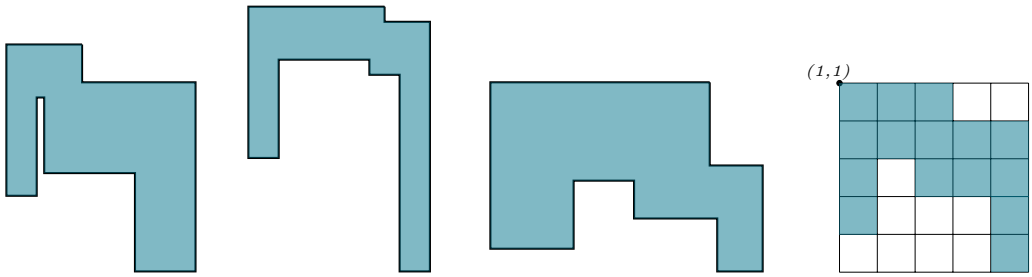


Figure 2.1: (a) Three 12-ogons in general position. (b) The corresponding grid 12-ogon.

If we assume that the northwest corner of the bounding square has coordinates $(1, 1)$, then the H-edges lie on the lines $y = 1,\ ...,\ y = \frac{n}{2}$ and the V-edges lie on the lines $x = 1,\ ...,\ x = \frac{n}{2}$. The grid ogon corresponding to P is found by mapping the edges of P to grid lines using the order in which they are found in a top-to-bottom (left-to-right) planar sweep. Conversely, as Figure 2.1 suggests, we can create many instances of n-ogons from a grid n-ogon by spacing the grid lines randomly, while keeping their relative order.

2.1.1 Inflate-Paste Algorithm

The *Inflate-Paste* algorithm [30] is an iterative method that creates an n-vertex grid ogon from a unit square in $O(n^2)$ time, using $O(n)$ space. At each iteration, it applies two transformations (called *Inflate* and *Paste*) to glue a new rectangle to the previous

grid ogon, yielding a new grid ogon with one more reflex vertex. Therefore, it performs $r = \frac{n-4}{2}$ iterations to produce a grid n-ogon. The rectangle is defined by a convex vertex v and a point in the exterior of the polygon. This point belongs to a region denoted by $FSN(v)$ that consists of points *rectangularly visible* to v, as we will see below. In Figure 2.2, we sketch the idea of the method.

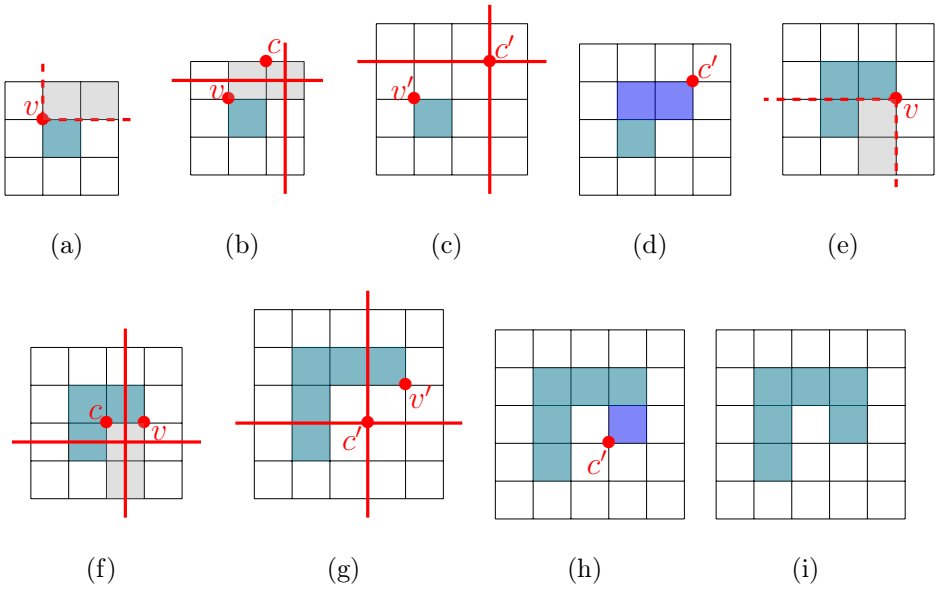


Figure 2.2: (a)(e) Select v and find $FSN(v)$ (• color). (b)(f) Select a cell in $FSN(v)$. (c)(g) Apply *Inflate*. (d)(h) *Paste* new rectangle. (i) Final grid 8-ogon.

Inflate is used to insert new gridlines, which are required to ensure that the polygon is in general position. *Paste* is used to glue a rectangle to an horizontal edge of the current grid ogon. The rectangle is fixed to a convex vertex v that is an extreme point of that edge. When we apply the method to create a polygon at random, in each iteration, a convex vertex v is selected randomly, as well as a rectangle to be glued. The sole restriction is that the rectangle belongs to the *free neighborhood* of v, i.e. to $FSN(v)$, which is computed also in that step.

$FSN(v)$ is a subset of the points that are in the exterior of the polygon and that are rectangularly visible to v. More accurately, $FSN(v)$ consists of the external points that are rectangularly visible from v in the quadrant with origin v that contains the horizontal edge $e_H(v)$ and the inversion of the vertical edge $e_V(v)$, incident to v. The inversion of $e_V(v)$ is its reflection with respect to v.

We say that a point p is rectangularly visible from v if there is an axis-aligned rectangle that contains v and p, but no point in the interior of P. For the definition of the free neighborhood, we assume that the grid is bounded by a square, as we show in Figure 2.2, and that $(0, 0)$ is its northwest corner. All the inflate-paste transformations keep these extra gridlines empty.

At the beginning of an iteration, the algorithm selects a convex vertex v of the grid ogon computed in the previous iteration and determines $FSN(v)$, as we illustrate in Figure 2.2. Then, it selects a cell in $FSN(v)$, at random. Its center and the vertex v define the opposite corners of the new rectangle to be glued by the Paste operation. But, firstly, it applies the Inflate operation to insert two new lines, a H-line and a V-line for the new edges that will meet at $c' = (c_x + 1, c_y + 1)$. Here, (c_x, c_y) is the northwest corner of the selected cell (this point is represented by c).

The Inflate procedure is rather simple: the y-coordinate of any H-line such that $y > c_y$ is incremented by 1; similarly, the x-coordinate of any V-line such that $x > c_x$ is increased by 1; for the remaining lines, nothing changes.

The Paste procedure distinguishes two situations to keep the polygon in CCW order. If $a_H(v')$ is after v' then it removes v' and inserts the chain $(v'_x, c'_y),\ c',\ (c'_x, v'_y)$ in its place. If $a_H(v')$ is before v' then it replaces v' by the chain $(c'_x, v'_y),\ c',\ (v'_x, c'_y)$. Here, v' refers to v after the shift carried out by Inflate, which may change its coordinates.

The computation of $FSN(v)$ is the more complex part of the method. Nevertheless, we can exploit the structural properties of the grid ogons and the fact that $FSN(v)$ is a *Ferrers diagram* to find a partition of $FSN(v)$ into rectangles in $O(n)$ time and space. The partition is useful to select and locate the grid cell in $O(n)$ time.

The algorithm is based on planar sweep, following the work by Overmars and Wood [26]. The horizontal sweeping line starts at $e_H(v)$ and moves towards the exterior of P, that is, in the direction given by the inversion of $e_V(v)$. This direction defines the quadrant that contains $FSN(v)$, as we mentioned above. The *sweep line status* keeps a *visibility interval*, which gives the current width of $FSN(v)$. In each event, we have to check whether there is some point that shrinks the visibility interval. In case there is, we have to update $FSN(v)$ and the current visibility interval.

The planar sweep stops, i.e, the algorithm terminates, when the visibility interval becomes empty or the sweeping line reaches the boundary of the grid. The result is a partition of $FSN(v)$ into rectangles, each one defined as $(y\text{-}start, x\text{-}interval, y\text{-}finish)$. $x\text{-}interval$ is the visibility interval of v between events $y = y\text{-}start$ and $y = y\text{-}finish$.

Figure 2.3 sketches a computation of $FSN(v)$.

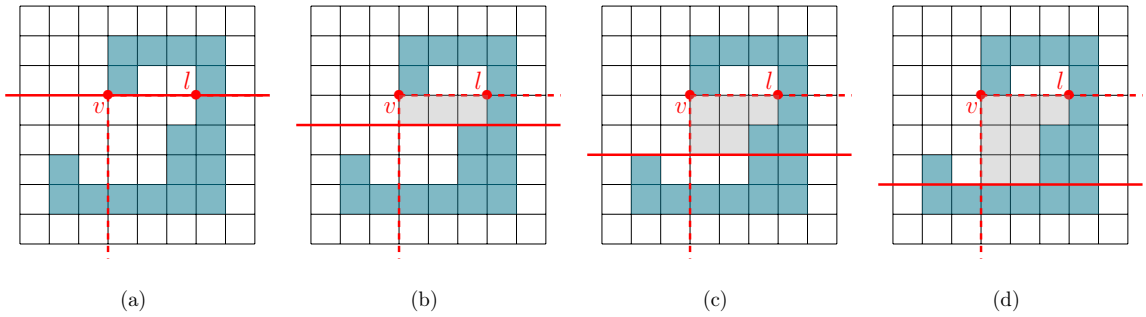


Figure 2.3: (a) Finding point l. Set of rectangles: (a) $\{(v_y,]v_x, l_x[, \_)\}$; (b) $\{(v_y,]v_x, l_x[, v_y + 1), (v_y + 1,]v_x, l_x - 1[, \_)\}$; (c) $\{(v_y,]v_x, l_x[, v_y + 1), (v_y + 1,]v_x, l_x - 1[, \_)\}$; (d) $\{(v_y,]v_x, l_x[, v_y + 1), (v_y + 1,]v_x, l_x - 1[, v_y + 3)\}$.

In the event at height y', the visibility interval is defined as $]x_1, x_2[$ if every point (x', y') with $x_1 \leq x' \leq x_2$ is rectangularly visible to v.

The first event is always at $y' = v_y$ and the sweep line status is $]min(v_x, l_x), max(v_x, l_x)[$. Here, l is a point with $l_y = v_y$ and belongs to the V-edge that cuts the rectangular visibility of v. In case there is no such V-edge, l is given by the correspondent V-line limit of the grid boundary. The first rectangle is initialized as $(v_y,]min(v_x, l_x), max(v_x, l_x)[, \_)$.

At each event, with the current rectangle $(y\text{-}start,]x_1, x_2[, \_)$, the extreme points of the H-edge at height y' are analyzed. If the extreme point q is such that $q_x \in]x_1, x_2[$ then the rectangle is completely processed, $(y\text{-}start,]x_1, x_2[, y')$, and a new one is created, $(y', x\text{-}interval, \_)$. The $x\text{-}interval$, as well as the sweep line status, is $]q_x, x_2[$ if $q_x > x_1$ and $]x_1, q_x[$ otherwise.

The proof of Inflate-Paste completeness [30] was done by observing the H-partition of the grid orthogonal polygons. Since the dual graph of $\Pi_H(P)$ is always a tree, it is possible to remove a leaf from it. If we do that and deflate the resulting polygon, we get a grid $(n-2)$-ogon P'. Therefore, assuming that P' is constructed by the method, P can be obtained in the next Inflate-Paste step.

2.2 Implementation

In this section, we describe in detail our generator for grid ogons, using pseudocode.

Algorithm 1 is the main function of the whole process. Given the number of vertices of the polygon to be generated, n, the algorithm performs r Inflate-Paste transformations. Table 2.1 shows the auxiliary functions used in the implementation.

Any vertex of the grid ogon is identified by an unique id. This and other details of the implementation such as the main data structures are explained below.

At each iteration, a convex vertex id v_id is picked from set C. The id of $a_H(v)$ and the interval of x-coordinate values of $e_H(v)$ are kept in the variables a_Hv_id and $e_Hv\_x\text{-}interval$ for future use. Then, $FSN(v)$ is computed, a cell is chosen at random from it and the Inflate-Paste process begins.

GetRandomCell chooses a cell from $FSN(v)$ region and returns its northwest corner $c = (c_x, c_y)$, in $O(n)$ time. The function takes a number m at random, that will identify the cell. We consider an implicit enumeration of the cells in $FSN(v)$. The cell with label 1 is the nearest cell to v and all the remaining ones are numbered by rows, starting from the row that contains $e_H(v)$ (see Figure 2.4(a)). Since $FSN(v)$ is decomposed into rectangles, the cell with label m can be found by analyzing the areas of the rectangles. If $FSN(v)$ has rectangles r_1, r_2, ..., r_k then r_i contains the cell m if the sum of the areas of r_1, ..., r_{i-1} is less than m and the sum of the areas of r_1, ..., r_i is greater than or equal to m. Once the rectangle r_i is located, m is updated to m', by subtracting the area of the rectangles skipped to m. Now, m' is between 1 and the area of r_i (see Figure 2.4(b)). Given $\triangle_H$, $\triangle_V$ and the new cell number m', it is possible to find (c_x, c_y) from the located rectangle in constant time.

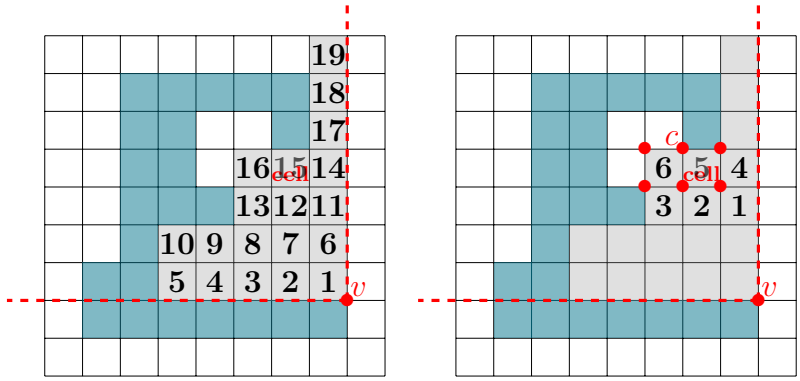
After the Inflate procedure, c becomes $c' = (c'_x, c'_y)$.

Algorithm 1 Inflate-Paste for grid n-ogons generation.

1: **procedure** GENERATEGRIDOGON(n)
2: INITSTRUCTURES();
3: $r \leftarrow (n-4)/2$;
4:
5: **while** $r \neq 0$ **do**
6: $v\_id \leftarrow$ GETRANDOMVERTEX(C);
7: $\{fsn, \triangle_H, \triangle_V\} \leftarrow$ GETFSN(v_id);
8: $(c_x, c_y) \leftarrow$ GETRANDOMCELL($fsn, \triangle_H, \triangle_V$);
9:
10: $(c'_x, c'_y) \leftarrow$ INFLATE((c_x, c_y));
11: $a_H v\_id \leftarrow$ GETADJACENT(v_id, H);
12: $v_y \leftarrow$ GETCOORDINATE(v_id, y); $e_H v\_x\text{-}interval \leftarrow$ GETINTERVALEDGE(v_y, H);
13: $\{c\_id, a_V c\_id\} \leftarrow$ PASTE($v\_id, a_H v\_id, (c'_x, c'_y)$);
14:
15: $C \leftarrow C \cup \{c\_id\}$;
16: **if** $c'_x \notin e_H v\_x\text{-}interval$ **then**
17: $C \leftarrow (C \setminus \{a_H v\_id\}) \cup \{a_V c\_id\}$;
18: **end if**
19:
20: $dim \leftarrow dim + 1$;
21: $r \leftarrow r - 1$;
22: **end while**
23: **end procedure**

Function	Description	Algorithms
GetRandomVertex(C)	Selects at random, a vertex id from set C.	1, 14
GetRandomCell($fsn, \triangle_H, \triangle_V$)	Returns the coordinates of the northwest corner c of a cell picked randomly from fsn.	1, 14
GetAdjacent(v_id, dir)	Returns the id of $a_{dir}(v)$, given v_id.	1, 13, 14, 16, 17, 18
GetCoordinate($v\_id, axis$)	Gives the v_{axis} coordinate, given v_id.	1, 2, 7, 12, 13, 14, 15, 16
GetIntervalEdge($coord, dir$)	If $0 < coord < dim + 1$, returns the interval of x-coordinate (y-coordinate) values that H-edge $y = coord$ (V-edge $x = coord$) has, for $dir = H$ ($dir = V$). Otherwise, the interval of the correspondent H-gridline (V-gridline) is returned.	1, 3, 4, 14
GetWay(v_id, dir)	Returns $\triangle_H = -\frac{a_V(v)_y - v_y}{\lvert a_V(v)_y - v_y \rvert}$ if $dir = H$ and $\triangle_V = \frac{a_H(v)_x - v_x}{\lvert a_H(v)_x - v_x \rvert}$ if $dir = V$, given v_id.	2, 12, 13, 15, 16
AddVertices($p_{start}, p_{fst\_new}, p_{snd\_new}$)	Inserts the ids p_{fst_new} and p_{snd_new} of the new vertices into the circular list P.	7

Table 2.1: Auxiliary functions for the implementation of the grid n-ogons generator.



(a) (b)

Figure 2.4: How cells are implicitly enumerated in $FSN(v)$. (a) Finding the rectangle that contains cell 15. (b) Finding the cell within the rectangle.

C is the set of convex vertices of the polygon, which are represented by their ids. In this way, we have direct access to any convex vertex. Once an Inflate-Paste step ends, C has to be updated. The point which represents the random cell in $FSN(v)$ is a new convex vertex and, thus, its id, c_id, is added to C. After the Inflate step, when $a_H(v')$ is convex but becomes the new reflex vertex once the rectangle is glued, $a_H(v')$ has to be replaced by $a_V(c')$ in C. This happens when c' surpasses $e_H(v')$, i.e., c'_x is not in $e_H v\_x\text{-}interval$. Otherwise, $a_V(c')$ is the new reflex vertex.

The global data structures used in our implementation were carefully chosen to simplify the most important and complex steps of the algorithm, namely the computation of $FSN(v)$ region. Table 2.2 presents these main structures and Figures 2.5 and 2.6 show a visual representation of a grid ogon with these structures.

Note that $line\_info_H[].coord$ is the inverse function of $grid_H[]$. It is easy to see that the main structures at Table 2.2 use linear space on the number of vertices.

grid	A matrix that provides information about the grid. $grid_H[i]$ ($grid_V[i]$) is the id of the horizontal (vertical) grid line $y = i$ ($x = i$).
line_info	A matrix that provides information about the grid ogon and the grid itself. $line\_info_H[i].coord$ ($line\_info_V[i].coord$) is the y-coordinate (x-coordinate) of the horizontal (vertical) grid line with id i; $line\_info_H[i].vertex$ ($line\_info_V[i].vertex$) is the id of a vertex that lies on the horizontal (vertical) grid line with id i.
P	A circular doubly linked list that represents the vertices of the polygon, in CCW. $P[i].line_H$ ($P[i].line_V$) is the id of the horizontal (vertical) grid line where the vertex with id i lies on; $P[i].next$ and $P[i].prev$ are the ids of the next and the previous vertices, respectively.
dim	Variable that has the current number vertices of the grid ogon divided by two. $(dim + 2) \times (dim + 2)$ are the current dimensions of the grid.
C	The ids of the convex vertices of the current grid ogon.

Table 2.2: Global data structures for the implementation of the grid n-ogons generator.

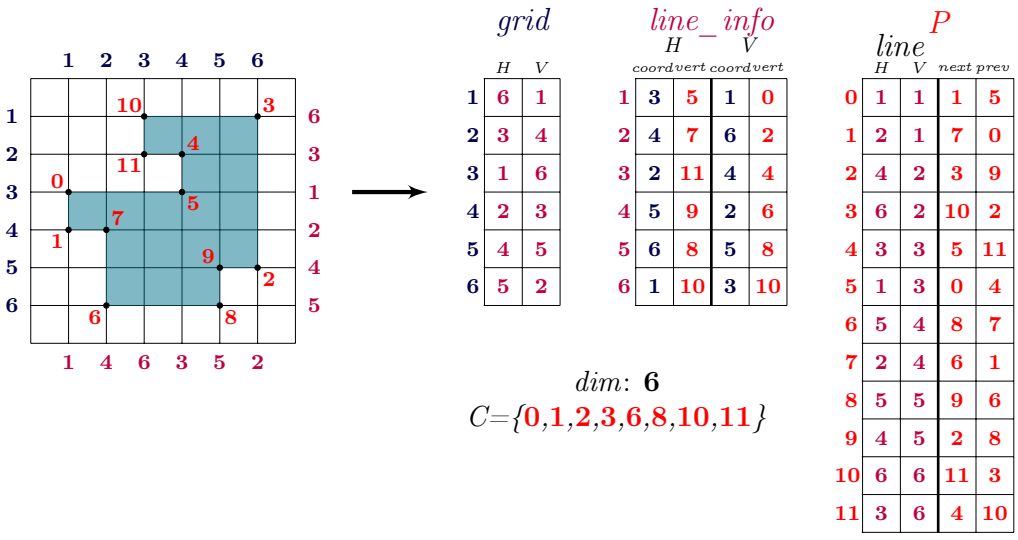


Figure 2.5: Global structures for a grid 12-ogon generated by the algorithm. Numbers with color: • are *grid* coordinates; • are *line_info* ids; • are P ids.

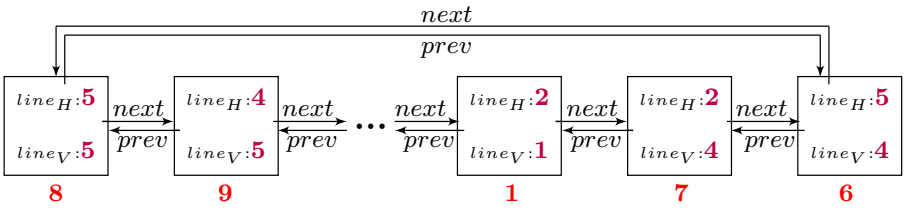


Figure 2.6: Circular list scheme for the structure P displayed in Figure 2.5.

$grid_H[]$ lets us make an ordered search in the relevant H-edges during the $FSN(v)$ computation. We have access to the H-edge with the same y-coordinate as the current position of the sweeping H-line, in constant time. Thus, the computation of $FSN(v)$ takes $O(n)$ time. Furthermore, $grid_V[]$ allows us to find the vertical line that defines the initial V-limit of $FSN(v)$ in $O(n)$ time as well, since the search for this bound is similar to a vertical line sweep.

$line\_info$ is the middle structure of grid ogon abstraction, linking P to the $grid$. It lets us to inflate P implicitly by inflating the grid explicitly (only $grid$ and $line\_info$ are updated).

In the Paste procedure, $grid$ is not modified, whereas the new convex and reflex vertices are added to $line\_info$ and P.

The Inflate and Paste take $O(n)$ and $O(1)$ time, respectively. Therefore, using the described data structures, the total time is $O(n^2)$.

InitStructures initializes all the global data structures, resulting into the unit square grid 4-ogon. For all $i \in \{1,2\}$ and $dir \in \{H,V\}$, we settle $grid_{dir}[i] = i$ and $line\_info_{dir}[i].coord = i$.

For P structure, we assign $P[i].line_H = \lfloor \frac{(i+1) \bmod 4}{2} \rfloor + 1$, $P[i].line_V = \lfloor \frac{i}{2} \rfloor + 1$, $P[i].prev = (i-1) \bmod 4$ and $P[i].next = (i+1) \bmod 4$, for each $i \in \{0,1,2,3\}$. We also add the id vertices 0 and 2 to the lines with ids 1 and 2, respectively. More formally, for $dir \in \{H,V\}$, we set $line\_info_{dir}[1].vertex = 0$ and $line\_info_{dir}[2].vertex = 2$.

Then, the set of convex vertices will be $C = \{0,1,2,3\}$ and the dimension $dim = 2$.

The $FSN(v)$ computation steps are organized in GetFSN function (Algorithm 2).

Here, the quadrant to which $FSN(v)$ belongs is expressed by $\triangle_H \in \{-1, 1\}$ and $\triangle_V \in \{-1, 1\}$. These variables allow us to move the sweeping lines without being worried about symmetries. $\triangle_H = -1$ ($\triangle_V = -1$) means that all cells in $FSN(v)$ have y-coordinate (x-coordinate) less or equal to v_y (v_x). $\triangle_H = 1$ ($\triangle_V = 1$) when all cells in $FSN(v)$ have y-coordinate (x-coordinate) greater or equal to v_y (v_x).

Algorithm 2 Computation of the free neighborhood of v.

```
1: function GetFSN(v_id)
2:     v_x ← GetCoordinate(v_id, x); v_y ← GetCoordinate(v_id, y);
3:     △_H ← GetWay(v_id, H); △_V ← GetWay(v_id, V);
4:
5:     l_x ← GetFSNx-Limit((v_x, v_y), △_H, △_V);
6:     fsn ← GetRectangles((v_x, v_y), l_x, △_H, △_V)
7:
8:     return {fsn, △_H, △_V}
9: end function
```

GetFSNx-Limit (Algorithm 3) returns the x-coordinate of the V-edge (or V-line) that bounds horizontally the rectangular visibility of v. The algorithm scans the V-edges (and the V-gridline if necessary), as if it were performing a sweep with a V-line, to find the first one that contains the V-segment defined by $y = v_y$ and $y = v_y + \triangle_H$.

Algorithm 3 Finding the interval visibility limit.

```
1: function GetFSNx-Limit((v_x, v_y), △_H, △_V)
2:     curr_x ← v_x + △_V;
3:     y-interval ← GetIntervalEdge(curr_x, V);
4:     while ¬(v_y ∈ y-interval ∧ (v_y + △_H) ∈ y-interval) do
5:         curr_x ← curr_x + △_V;
6:         y-interval ← GetIntervalEdge(curr_x, V);
7:     end while
8:
9:     return curr_x
10: end function
```

GetRectangles (Algorithm 4) returns $FSN(v)$ decomposed into rectangles, which are stored at fsn set. The implementation follows directly from the algorithm presented at Section 2.1.1.

Algorithm 4 The planar sweep algorithm for finding the free neighborhood.

```
1: function GetRectangles((s_x, s_y), l_x, △_H, △_V)
2:     fsn ← ∅;
3:     r_y-start ← s_y;
4:     r_x-interval ← ]min(s_x, l_x), max(s_x, l_x)[;
5:
6:     curr_y ← s_y + △_H;
7:     x-interval ← GetIntervalEdge(curr_y, H);
8:     while s_x ∉ x-interval do
9:         if x-interval ∩ r_x-interval ≠ ∅ then
10:            r_y-finish ← curr_y;
11:            fsn ← fsn ∪ {r};
12:            r_y-start ← curr_y;
13:            if △_V < 0 then
14:                r_x-interval ← ]max(x-interval), max(r_x-interval)[;
15:            else
16:                r_x-interval ← ]min(r_x-interval), min(x-interval)[;
17:            end if
18:        end if
19:        curr_y ← curr_y + △_H;
20:        x-interval ← GetIntervalEdge(curr_y, H);
21:    end while
22:
23:    r_y-finish ← curr_y;
24:    fsn ← fsn ∪ {r};
25:
26:    return fsn
27: end function
```

The (s_x, s_y) parameter is the starting point for finding $FSN(v)$, which is exactly (v_x, v_y) in the general case. The variable l_x gives the x-coordinate that limits the rectangular visibility of v. For an interval I, the values $min(I)$ and $max(I)$ just give the minimum and maximum values defining I, respectively.

INFLATE (Algorithm 5) gathers the Inflate operations.

Algorithm 5 The *Inflate* procedure.

```
1: function INFLATE((c_x, c_y))
2:     INFLATEAXIS(c_y, H);
3:     INFLATEAXIS(c_x, V);
4:     return (c_x + 1, c_y + 1)
5: end function
```

INFLATEAXIS (Algorithm 6) inflates the y-coordinates (x-coordinates) of the grid lines if $dir = H$ ($dir = V$). The ids of the new H-line and V-line in the grid are $dim + 1$.

Algorithm 6 The *InflateAxis* operation.

```
1: function INFLATEAXIS(coord, dir)
2:     for i = dim until coord + 1 do
3:         grid_dir[i + 1] ← grid_dir[i];
4:         line_info_dir[grid_dir[i]].coord ← i + 1;
5:     end for
6:     grid_dir[coord + 1] ← dim + 1;
7:     line_info_dir[dim + 1].coord ← coord + 1;
8: end function
```

PASTE (Algorithm 7) adds the two new vertices, c and $a_V c$, to $line\_info$ and P data structures. The function returns the ids of the new vertices to quickly update the set C afterwards.

Algorithm 7 The *Paste* operation.

1: **function** PASTE($v\_id, a_H v\_id, (c'_x, c'_y)$)
2: $c\_id \leftarrow 2 \times dim;\ a_V c\_id \leftarrow\ 2 \times dim + 1;$
3:
4: $line\_info_H[grid_H[c'_y]].vertex \leftarrow c\_id;\ line\_info_V[grid_V[c'_x]].vertex \leftarrow c\_id;$
5: $P[c\_id].line_H \leftarrow grid_H[c'_y];\ P[c\_id].line_V \leftarrow grid_V[c'_x];$
6:
7: $v_y \leftarrow$ GETCOORDINATE(v_id, y);
8: $line\_info_H[grid_H[v_y]].vertex \leftarrow a_V c\_id;$
9: $P[a_V c\_id].line_H \leftarrow grid_H[v_y];\ P[a_V c\_id].line_V \leftarrow grid_V[c'_x];$
10:
11: $P[v\_id].line_H \leftarrow grid_H[c'_y];$
12:
13: **if** $P[v\_id].next = a_H v\_id$ **then**
14: ADDVERTICES($v\_id, c\_id, a_V c\_id$);
15: **else**
16: ADDVERTICES($a_H v\_id, a_V c\_id, c\_id$);
17: **end if**
18:
19: **return** $\{c\_id, a_V c\_id\}$
20: **end function**

After the Paste procedure, v is no longer a vertex of the polygon. However, to simplify the changes, v's id is used to represent the new convex vertex adjacent to $a_V(v)$. Thus, only $e_H(v)$ has to be updated, i.e., we modify the value of $line\_info_H[grid_H[v_y]].vertex$.

ADDVERTICES is a function that inserts the chain of ids of the new vertices, p_{fst_new} and p_{snd_new}, into the circular list P, given the id p_{start} (id of a vertex that was already in P and identifies the position where the new chain will be added). The if-else clause in Algorithm 7 assures that the CCW order of P is preserved. Figure 2.7 shows the circular list of P from Figure 2.6 modified after the next Inflate-Paste step, where the convex vertex id selected from set C was 1.

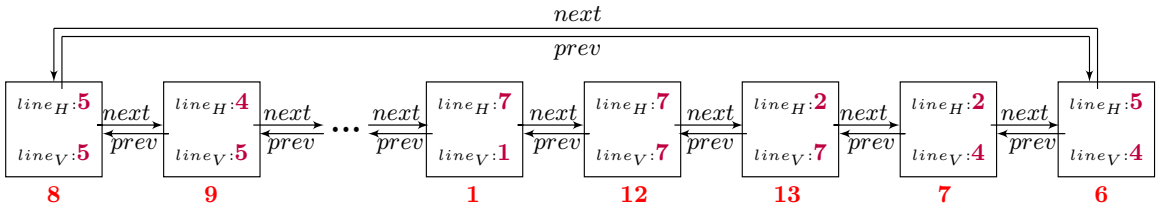


Figure 2.7: The new circular list from Figure 2.6 after calling AddVertices$(1, 12, 13)$.

2.3 Generation of Generic n-Ogons

As we observed in section 2.1, any n-ogon can be mapped into a n-ogon in general position and, on the other hand, there is exactly one grid n-ogon that matches the latter. Hence, once a grid n-ogon is created, a n-ogon can be obtained by moving (stretching) the edges of the grid n-ogon.

The algorithm for creating n-ogons is shown in pseudocode as Algorithm 8.

Algorithm 8 Generation of a generic n-ogon from a grid n-ogon.

```
1: procedure GenerateOgon(n, maxCoord, cProb, sTimes)
2:     GenerateGridOgon(n);
3:
4:     InitDefaultStretcher();
5:     StretchGridOgon(maxCoord, cProb, sTimes);
6: end procedure
```

The idea is to create a grid n-ogon and then randomly picking and stretching edges of the grid ogon several times to end up with a generic n-ogon. The vertices of the final n-ogon have always positive integer coordinates.

The new global data structure *stretched* is used to record the new coordinates. $stretched_y[i]$ gives the current position of the i-th horizontal edge and $stretched_x[i]$ gives the current position of the i-th vertical edge. This structure is useful to ensure that the relative order of the edges is preserved. Clearly, initially, $stretched_x[i]$ and $stretched_y[i]$ are equal to i, for all $1 \leq i \leq dim$, meaning that the n-ogon is the grid n-ogon itself before any stretching. The function InitDefaultStretcher does this simple initialization.

The parameters $maxCoord$, $cProb$ and $sTimes$ (Table 2.3) allow us to have some control on the stretching phase.

$maxCoord$	$maxCoord_x$ ($maxCoord_y$) sets the maximum x-coordinate (y-coordinate) for the V-edges (H-edges) of the polygon.
$cProb$	The probability of making the target edge collinear with an edge in the following line, if that is possible.
$sTimes$	The number of edge translations to be performed.

Table 2.3: Parameters of the GENERATEOGON procedure.

The parameter $sTimes$ should be large enough. Indeed, let the i-th vertical edge of the grid n-ogon be the one in the line $x = i$, for $1 \leq i \leq dim$. In the same way, let the i-th horizontal edge be the one in the line $y = i$. If the $(dim - 1)$-th H-edge has just been translated and $stretched_y[dim-1] = \lfloor \frac{maxCoord_y}{2} \rfloor$ then the first $(dim-2)$ H-edges would be confined to a rectangle of height $\lfloor \frac{maxCoord_y}{2} \rfloor$. Moreover, when we have these conditions and the dim-th H-edge has $stretched_y[dim] = maxCoord_y$, the unbalance of this polygon can be notorious. Therefore, $sTimes$ should be large enough for an edge to have chances to be translated more than few times, as well as the preceding ones.

If $cProb = 0$, the resulting n-ogon will be in general position. Otherwise, the value $(cProb \times 100)\%$ defines the probability that the target edge becomes collinear with another edge. More accurately, if the target edge is the i-th horizontal edge, for instance, then this value is the probability that the edge is moved to the line $y = stretched_y[i + 1]$, provided it can be moved there and without intersecting any other edge. In a subsequent iteration, it can be moved to a different line.

The function STRETCHGRIDOGON is presented as Algorithm 9.

Algorithm 9 Main function to stretch the grid ogon.

1: **function** StretchGridOgon($maxCoord, cProb, sTimes$)
2: $ledgeCoord_x \leftarrow 0$; $ledgeCoord_y \leftarrow 0$;
3: SetMaxStretchUpper($\lfloor \frac{maxCoord_x - dim}{dim} \rfloor, x$); SetMaxStretchUpper($\lfloor \frac{maxCoord_y - dim}{dim} \rfloor, y$);
4:
5: **for** $i = 1$ *until* $sTimes$ **do**
6: $axis \leftarrow$ GetRandomAxis(); $edgeCoord \leftarrow$ GetRandomEdge();
7:
8: **if** $edgeCoord < ledgeCoord_{axis}$ **then**
9: $maxStretch \leftarrow$ GetMaxStretchLower($axis$);
10: **else**
11: $maxStretch \leftarrow$ GetMaxStretchUpper($axis$);
12: **end if**
13: TranslateEdge($edgeCoord, maxStretch, maxCoord, cProb, axis$);
14:
15: $ledgeCoord_{axis} \leftarrow edgeCoord$;
16: **if** $edgeCoord > 1$ **then**
17: SetMaxStretchLower($\lfloor \frac{stretched_{axis}[edgeCoord]}{edgeCoord-1} \rfloor, axis$);
18: **end if**
19: SetMaxStretchUpper($\lfloor \frac{maxCoord_{axis} - stretched_{axis}[edgeCoord]}{dim-edgeCoord+1} \rfloor, axis$);
20: **end for**
21: **end function**

The edge to be moved is selected randomly and, after a move, the current bounds for the translations in the same direction are updated. In the implementation we try to balance the current space available for future translations. At start, the algorithm fixes the upper bound for translation as $\lfloor \frac{maxCoord_x - dim}{dim} \rfloor$, for every V-edge (and $\lfloor \frac{maxCoord_y - dim}{dim} \rfloor$, for every H-edge). If the i-th vertical edge is translated, $stretched_x[i]$ changes, and the bounds for the first $i-1$ vertical edges are updated to $\lfloor \frac{stretched_x[i]}{i-1} \rfloor$. In addition, the bounds for the last $dim - i + 1$ vertical edges change to $\lfloor \frac{maxCoord_x - stretched_x[i]}{dim-i+1} \rfloor$. We proceed in a similar way if the target edge was the i-th horizontal edge (using $stretched_y[i]$ and $maxCoord_y$ in the formula, instead).

The functions GetRandomAxis and GetRandomEdge choose the next edge to be translated, at random. The first one returns x or y, which defines the direction of the edge, and the other returns an integer between 1 and dim, which identifies the edge.

The function TranslateEdge performs the translation. The parameters $edgeCoord$ and $axis$ identify the edge to be moved, and $maxStretch$ defines the upper bound for the difference between the new position and the current one. TranslateEdge is presented in pseudocode as Algorithm 10. When the new position is defined by the assignment in line 17, the edge would be collinear with the subsequent edge.

Algorithm 10 Function to translate an edge.

```
1:  function TranslateEdge(edgeCoord, maxStretch, maxCoord, cProb, axis)
2:      if edgeCoord = dim then
3:          maxCoord_dist ← maxCoord_axis − stretched_axis[edgeCoord];
4:          stretch ← GetRandomInteger(0, min(maxStretch, maxCoord_dist));
5:          stretched_axis[edgeCoord] ← stretched_axis[edgeCoord] + stretch;
6:          return
7:      end if
8:
9:      upedge_dist ← stretched_axis[edgeCoord + 1] − stretched_axis[edgeCoord];
10:     if maxStretch < upedge_dist then
11:         stretch ← GetRandomInteger(0, maxStretch);
12:     else if EdgeCouldIntersect(edgeCoord, upedge_dist, axis) then
13:         stretch ← GetRandomInteger(0, max(0, upedge_dist − 1));
14:     else if GetRandomReal(0, 1)> cProb then
15:         stretch ← GetRandomInteger(0, max(0, upedge_dist − 1));
16:     else
17:         stretch ← upedge_dist;
18:     end if
19:
20:     stretched_axis[edgeCoord] ← stretched_axis[edgeCoord] + stretch;
21: end function
```

The function ensures that the edge does not surpass nor intersect any subsequent edges. It calls EdgeCouldIntersect to check whether the edge defined by *edgeCoord* and *axis* could intersect some edge on the line $stretched_{axis}[edgeCoord + 1]$. This function is presented as Algorithm 11.

Algorithm 11 Verifying if the edge may intersect subsequent edges after stretching.

```
1:  function EdgeCouldIntersect(edgeCoord, maxStretch, axis)
2:      dir ← GetDirection(axis);
3:
4:      for i = edgeCoord + 1 until dim do
5:          if stretched_axis[edgeCoord] + maxStretch < stretched_axis[i] then
6:              return false
7:          end if
8:
9:          edge_axis-interval ← GetIntervalEdgeOgon(edgeCoord, dir);
10:         upedge_axis-interval ← GetIntervalEdgeOgon(i, dir);
11:         if edge_axis-interval ∩ upedge_axis-interval ≠ ∅ then
12:             return true
13:         end if
14:     end for
15:
16:     return false
17: end function
```

GetRandomInteger(min_val, max_val) is an auxiliary function and returns a random integer between min_val and max_val. Similarly, GetRandomReal(min_val, max_val) returns a random real number in that range. The other relevant auxiliary functions are described in Table 2.4.

In Appendix A, we show instances of grid 24-ogons (both generic and from the studied subclasses) that were created by our generators, as well as a 24-ogon derived from each of them using the stretching algorithm with $maxCoord_x = maxCoord_y = 48$, $cProb = 0.5$ and $sTimes = 48$.

Function	Description	Algorithms
SetMaxStretchLower($maxStretch, axis$)^L SetMaxStretchUpper($maxStretch, axis$)^U	Saves the maximum stretching $maxStretch$ for the H-edges with y-coordinate less than^L (greater than or equal to^U) the y-coordinate of the last translated H-edge, if $axis = y$. The same for V-edges, w.r.t the x-coordinate if $axis = x$.	9
GetMaxStretchLower($axis$)^L GetMaxStretchUpper($axis$)^U	Retrieves the maximum stretching of the H-edges with y-coordinate less than^L (greater than or equal to^U) the y-coordinate of the last translated H-edge, if $axis = y$. The same for V-edges, w.r.t the x-coordinate if $axis = x$.	9
GetDirection($axis$)	Returns H if $axis = y$; V if $axis = x$.	11
GetIntervalEdgeOgon($coord, dir$)	Returns the interval of x-coordinate (y-coordinate) values that H-edge $y = coord$ (V-edge $x = coord$) has in $stretched$, for $dir = H$ ($dir = V$).	11

Table 2.4: Auxiliary functions for the implementation of the n-ogon generator.

Chapter 3

Creating Subclasses of Grid n-Ogons

In this chapter, we present tailored versions of the *Inflate-Paste* algorithm to generate grid ogons with specific properties, namely row-convex, convex, thin and spiral grid ogons.

The algorithm developed for the convex grid ogons is based on the one designed for the row convex family and the algorithm designed for the spiral grid ogons is based on the one developed for the thin family. In each section, we consider a class and state the relevant properties, and then describe the customized version of the algorithm.

3.1 Row-Convex Grid n-Ogons

Any column-convex grid ogon can be obtained from a row-convex grid ogon by performing a rotation of $\frac{\pi}{2}$ radians around the center of its bounding square, and reciprocally. Since the customization of the Inflate-Paste method to produce row-convex grid ogons is much more simple, we focused on their generation, instead of on the generation of column-convex grid ogons.

The algorithm is pretty similar to the original version of Inflate-Paste. The main difference is that only four convex vertices can be used for the expansion at each iteration: the two topmost and the two bottommost convex vertices of the current polygon. That is, these four vertices are the ones that lie on the H-edges that define the minimum bounding square. We can discard all the remaining convex vertices,

which we call the *internal convex vertices*, because if they were selected the polygon could not be row-convex anymore. Proposition 1 states this result. In Figure 3.1, we show an example that provides some intuition for the proof.

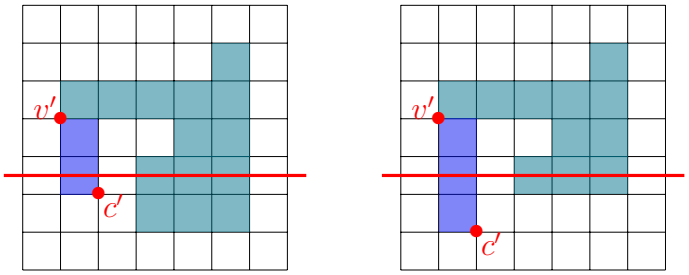


Figure 3.1: For all possible points c', i.e., for all the possible rectangles that could be glued at the internal vertex v' (color •), the resulting grid 12-ogon will not be row-convex.

Proposition 1. *Let P be a row-convex grid ogon, with $n \geq 6$ vertices, and let v be any internal convex vertex of P. If we use v to perform one more Inflate-Paste transformation, the grid $(n+2)$-ogon we obtain is not row-convex.*

Proof. In the proof, we use the notations introduced in the description of Inflate-Paste. The grid n-ogon P has an horizontal edge for each y such that $1 \leq y \leq \frac{n}{2}$. Since v is an internal convex vertex, we have $2 \leq v_y \leq \frac{n}{2} - 1$.

When we apply Inflate, the internal vertices are mapped to internal vertices, because the order induced by an horizontal planar sweep is preserved by this operation. This means that if v_y is incremented, then all H-edges below v are shifted downwards as well. If v_y is not incremented, then none of the H-edges above v moves. So, if v is an internal vertex then its image v' is an internal vertex of the inflated polygon, and reciprocally.

Now, let us assume, without loss of generality, that $y \geq v_y$ for all points in $FSN(v)$. Indeed all the other cases can be reduced to this one by symmetry. Then, for v' and c' we have $2 \leq v'_y \leq \frac{n}{2}$ and $3 \leq c'_y = c_y + 1 \leq \frac{n}{2} + 1$, and $c'_y - v'_y \geq 1$. It follows that, when we glue the rectangle defined by v' and c', the intersection of the polyomino with the row defined by the H-lines $y = c'_y - 1$ and $y = c'_y$ is not connected. Therefore, whichever cell we select in $FSN(v)$, the grid $(n+2)$-ogon we obtain is not row-convex. □

As a result, the set C of *active convex vertices* consists of the two topmost and the two bottommost vertices. Any of these vertices can be used for expansion, because no H-

line that crosses $FSN(v)$, for $v \in C$, intersects the interior of the current polyomino. Since $FSN(v)$ consists of a unique rectangle, with unit height, we can simplify the functions GetFSN and GetRandomCell.

The version of GetFSN we designed for the generation of a row-convex grid ogons is shown below and referred to as Algorithm 12.

Algorithm 12 Customized version of GetFSN to create row-convex grid ogons.

```
1:  function GetFSN(v_id)
2:      v_x ← GetCoordinate(v_id, x); v_y ← GetCoordinate(v_id, y);
3:      △_H ← GetWay(v_id, H); △_V ← GetWay(v_id, V);
4:
5:      r_{y-start} ← v_y;
6:      if △_V < 0 then
7:          r_{x-interval} ← ]0, v_x[;
8:      else
9:          r_{x-interval} ← ]v_x, dim + 1[;
10:     end if
11:     r_{y-finish} ← v_y + △_H;
12:     fsn ← {r};
13:
14:     return {fsn, △_H, △_V}
15: end function
```

The main function GenerateGridOgon is then adapted in the following way. We remove line 12 of Algorithm 1 and replace lines 15−18 by the instruction

$$C \leftarrow (C \setminus \{a_H v\_id\}) \cup \{c\_id\}.$$

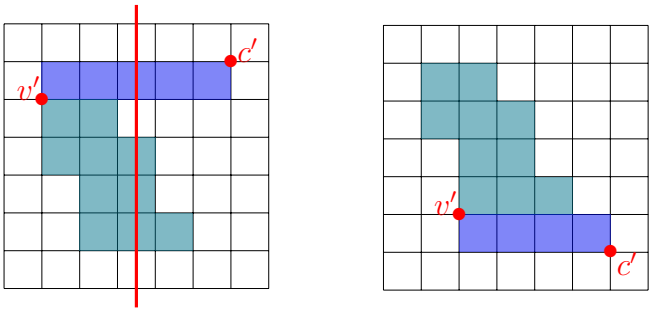
Finally, the function GetRandomCell is simplified, as the cell can be found in constant time now.

3.2 Convex Grid n-Ogons

By definition, every convex grid n-ogon has to be row-convex. The set C of the active convex vertices in each iteration will consist of the two bottommost and the two topmost

vertices of the polygon, again. Nevertheless, to ensure that the resulting polygon is also column-convex, we need to restrict the definition of $FSN(v)$.

Considering the way in which the rectangles are glued in Inflate-Paste, we may conclude that a non convex grid ogon cannot be used to produce a convex one. Now, if the grid n-ogon P is convex, and v is the vertex in C we select for expansion in iteration $(r+1)$, then all cells in the range defined by the edge $e_H(v)$ can be safely used. However, as the example in Figure 3.2 shows, we cannot select cells beyond $e_H(v)$, unless the other extreme point of $e_H(v)$, that is the vertex $a_H(v)$, belongs to line $x = 1$ or to line $x = \frac{n}{2}$.



(a) (b)

Figure 3.2: Two grid 12-ogons generated by the same convex grid 10-ogon. (a) The grid 12-ogon is not column-convex. (b) The grid 12-ogon is column-convex.

Indeed, since the grid orthogonal polygon has one edge in every line of its minimum bounding square, we know that when the x-coordinate of the vertex $a_H(v)$ satisfies $2 \leq x \leq \frac{n}{2} - 1$, then P has some V-edge to the left (right) of the edge $e_V(a_H(v))$ if $a_H(v)$ is to the left (right) of $e_V(v)$. Thus, if the rectangle we glue to obtain a new polygon grid $(n+2)$-ogon P' goes beyond $e_V(a_H(v))$, there exists some V-line L between $a_H(v)_x$ and c'_x such that $P' \cap L$ is not connected (see Figure 3.2(a)).

In case $a_H(v)$ belongs to the line $x = 1$, it is safe to include the cell whose northwest corner lies on the V-line $x = 0$, because it belongs to the empty column. Similarly, if $a_H(v)$ belongs to the line $x = \frac{n}{2}$, we can select the cell whose northwest corner lies on the V-line $x = \frac{n}{2}$.

This leads to a new version of the function GetFSN, that we present as Algorithm 13. This function is the unique part of the algorithm for the row-convex grid ogons we had to adapt to create convex grid ogons.

Algorithm 13 Customized version of GetFSN to create convex grid ogons.

```
1: function GetFSN(v_id)
2:     v_x ← GetCoordinate(v_id, x); v_y ← GetCoordinate(v_id, y);
3:     △_H ← GetWay(v_id, H); △_V ← GetWay(v_id, V);
4:     a_H v_id ←GetAdjacent(v_id, H); a_H v_x ← GetCoordinate(a_H v_id, x);
5:
6:     r_{y-start} ← v_y;
7:     if a_H v_x = 1 or a_H v_x = dim then
8:         r_{x-interval} ← ]min(v_x, a_H v_x + △_V), max(v_x, a_H v_x + △_V)[;
9:     else
10:        r_{x-interval} ← ]min(v_x, a_H v_x), max(v_x, a_H v_x)[;
11:    end if
12:    r_{y-finish} ← v_y + △_H;
13:    fsn ← {r};
14:
15:    return {fsn, △_H, △_V}
16: end function
```

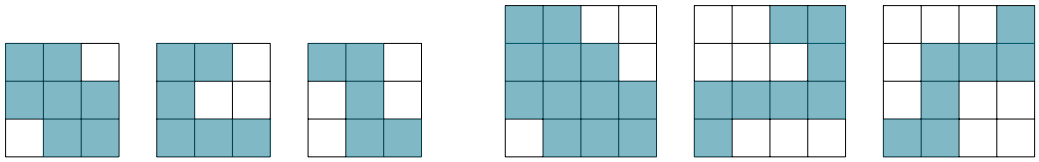
3.3 Thin Grid n-Ogons

The *thin* and the *fat* grid ogons were defined in [2] as the grid ogons for which the number of pieces of the rectilinear partition Π_{HV} is minimum and maximum, respectively. For the fat grid ogons with r reflex vertices we have

$$|\Pi_{HV}(P)| = \begin{cases} \frac{3r^2+6r+4}{4} & \text{if } r \text{ is even} \\ \frac{3(r+1)^2}{4} & \text{if } r \text{ is odd} \end{cases} .$$

whereas for the thin grid ogons the number of pieces of $\Pi_{HV}(P)$ is $2r + 1$. There is a unique fat grid n-ogon, up to symmetry [2], and its pattern is quite simple (see Figure 3.3).

This makes the generation of the fat class straightforward and less interesting than the generation of thin grid n-ogons, which is the one we address in this section.



(a) (b)

Figure 3.3: The fat and two thin grid n-ogons: (a) $n = 8$, $r = 2$ (b) $n = 10$, $r = 3$.

The thin grid ogons are the ones for which there are no intersection points between the horizontal and vertical cuts in the interior of the polygon. Moreover, for $n \geq 6$, the dual graph of $\Pi_{HV}(P)$ is a path graph.

We exploit these structural properties to adapt the Inflate-Paste method. As for the generation of row-convex and convex grid ogons, thin grid ogons can only be obtained from thin grid ogons and only a proper subset C of the convex vertices are active in each iteration, for $n \geq 6$. Actually, the following lemma and proposition are shown in [23].

Lemma 2. *Let P be a thin $(n+2)$-ogon. Then every grid n-ogon that yields P by Inflate-Paste is also a thin grid ogon.*

Proposition 2. *The unique convex vertices of a thin grid n-ogon P that can be used to yield a thin grid $(n+2)$-ogon, by Inflate-Paste, are the four convex vertices that belong to the r-pieces associated to the extreme points of the dual graph of $\Pi_{HV}(P)$ (which is a path graph).*

In contrast to the procedure described in [23], which adopts the original definition of $FSN(v)$ and, in the end, discards the polygon if it is not a thin polygon, the outcome of our algorithm is always a thin grid ogon. For that purpose, we need to restrict $FSN(v)$, as well. Essentially, we have to avoid all cases where the new cuts would intersect the previous ones, as we see in Figure 3.4.

We know that no intersection point between two cuts can be destroyed by Inflate, because this operation keeps the structure of the rectilinear partition (although the size of some pieces may increase, the dual graph does not change). In addition, no two vertical (or horizontal) cuts can overlap because, by definition, a grid ogon has no collinear edges. Therefore, when a new reflex vertex is added by Paste, the V-

cut (H-cut) it adds either intersects some previous H-cut (V-cut), creating internal intersection points, or does not intersect any previous cut.

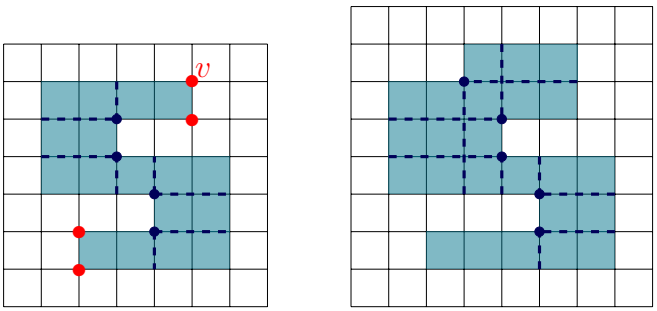


Figure 3.4: (a) A thin grid 12-ogon. (b) A grid 14-ogon arising from (a) that is not thin.

Now, we focus on the generation of the thin grid ogons by Inflate-Paste. By case analysis, we concluded that, for $n \geq 6$, we have to distinguish the case where the two active convex vertices p and q that belong to the extreme piece define a vertical edge (called *Type I*) from the case where they define an horizontal edge (called *Type II*), as we show in Figure 3.5.

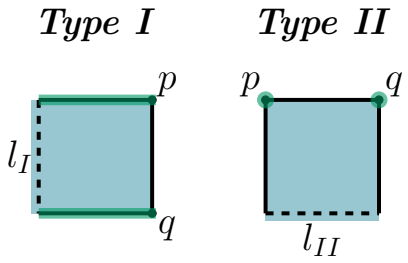


Figure 3.5: The two cases we must distinguish to restrict FSN (up to symmetry).

In case Type I, when we select p or q, any rectangle that does not surpass the vertical cut l_I can be glued. Otherwise, the horizontal cut determined by the new reflex vertex intersects that cut. In case Type II, when we select p, only the rectangles that surpass q can be glued and, similarly, when we select q, the rectangle must surpass p. Otherwise, the vertical cut incident to the new reflex vertex intersects the horizontal cut denoted by l_{II}, and the outcome cannot be a thin grid ogon.

We state this result as Proposition 3 and prove it by induction. The proof allows us to understand the structure of the thin grid ogons and at the same time shows how the Inflate-Paste method works for this subclass.

Proposition 3. *The outcome of the Inflate-Paste method is a thin grid ogon if and only if, in each iteration (other than the first one), we select a convex vertex v that belongs to a piece defining an extreme point of the dual graph of Π_{HV} and we follow the rules given for Type I and Type II to select a cell in $FSN(v)$.*

Proof. We proceed by induction on the number of reflex vertices r. Along the proof, P_r denotes a grid ogon with r reflex vertices.

Base case: In case $r \leq 1$, we do not need to impose any restriction, because all the grid ogons with $n \leq 6$ vertices are thin grid ogons. Up to an horizontal or vertical flip, when we apply Inflate-Paste to P_0 using a convex vertex v to obtain a grid ogon P_1, the situation is as sketched in Figure 3.6.

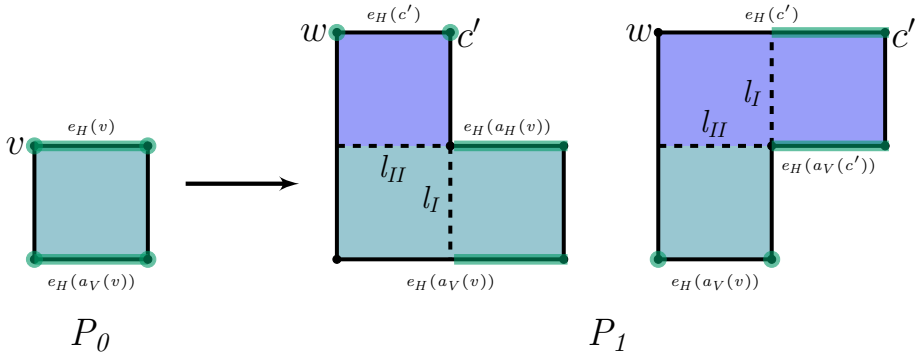


Figure 3.6: The thin grid 6-ogons, P_1, arising from the grid 4-ogon P_0 (up to symmetry).

We see that the rectilinear partition $\Pi_{HV}(P_1)$ consists of three pieces, its dual is a path graph and one of the extreme pieces is of Type I and the other of Type II. The two cuts l_I and l_{II} result from the extension of the edges that are incident to the reflex vertex.

If P_1 is as depicted on the left, the convex vertex $a_V(v)$ (adjacent to v in P_0) must be removed from the current set of active convex vertices C. Indeed, if $a_V(v)$ was used to generate a polygon P_r, for $r \geq 2$, then the new cuts will intersect l_I or l_{II}, for every rectangle in its free neighborhood. Similarly, if P_1 is the polygon on the right, we

cannot include the vertex w in the set C. In each case, we can use any of the remaining four convex vertices and, as we can see, they belong to extreme pieces of the chain.

In both cases, the extreme piece of Type I contains the horizontal edge that is incident to the reflex vertex of P_1. If the selected convex vertex is the other extreme point of that edge ($a_H(v)$ on the left, $a_V(c')$ on the right), there are no new restrictions. On the other hand, if we selected the other convex vertex in the extreme piece of Type I (i.e., c' in the right or the vertex that is vertically adjacent to $a_H(v)$ in the left), then we can glue a rectangle in the free neighborhood of this vertex if and only if it does not surpass l_I. Otherwise, the vertical cut added by the new reflex vertex intersects l_{II}.

As regards the other extreme piece, which is of Type II, if the selected vertex belongs to that piece, the rectangles that can be glued are the ones in the free neighborhood of such vertex that do not contain a vertical line that intersects l_{II}.

So, for P_1, the condition stated in the Proposition holds.

Induction step: We are going to show that if the condition holds for all thin grid ogons P_r, then it holds for all thin grid ogons P_{r+1}, for $r \geq 1$.

We know that the original Inflate-Paste method is complete, which means that every grid ogon with $r+1$ reflex vertices can be obtained from some grid ogon with r reflex vertices by gluing a rectangle using the Inflate-Paste transformation. Therefore, the hypothesis implies that any thin grid ogon P_{r+1} can be created and its predecessor P_r is a thin grid ogon. So, the proof reduces to showing that the conditions on the active convex vertices and on the types of the extreme pieces hold for P_{r+1} in next iteration. Since the Inflate operation preserves the structure of $\Pi_{HV}(P_r)$, the hypothesis still holds for the inflated polygon.

Hence, let us consider the case where the convex vertex v we select in P_r belongs to a Type I extreme piece.

Considering the induction hypothesis, the local structure of P_{r+1} will be of the form described in Figure 3.7, up to symmetries.

We can see that the extreme piece is replaced by a chain of three pieces. The new vertical cut divided that extreme piece in two and the rectangle we glued to obtain

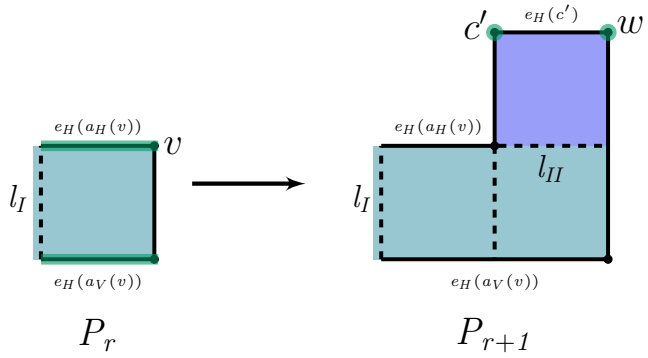


Figure 3.7: When v belongs to a piece of Type I in P_r, the new rectangle yields a piece of Type II in P_{r+1}.

P_{r+1} becomes the new extreme piece, which is of Type II. The vertex $a_V(v)$ has to be removed from the set of active convex vertices, and c' and w are added to the set. We can justify this claim as we did for P_1, in the base case (Type II).

The remaining structure of P_r and $\Pi_{HV}(P_r)$ is preserved, because the new cuts only affect the extreme piece of P_r that contains v. Hence, the other extreme piece of $\Pi_{HV}(P_{r+1})$ is the other extreme piece of P_r (possibly inflated). Therefore, the conditions hold if v belongs to a piece of Type I.

Now we show that the conditions hold also when the convex vertex v we select in P_r belongs to a Type II extreme piece. Under the induction hypothesis, the local structure of P_{r+1} will be of the form sketched in Figure 3.8, up to symmetries.

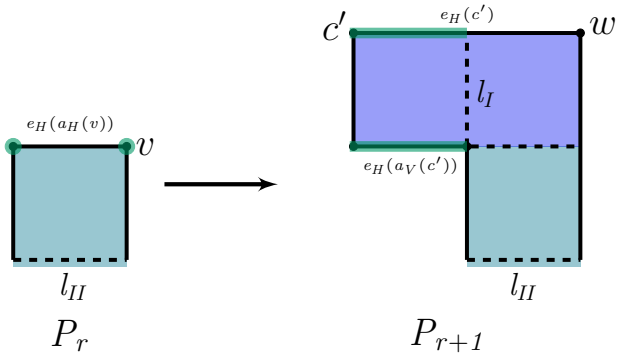


Figure 3.8: When v belongs to a piece of Type II in P_r, the new rectangle yields a piece of Type I in P_{r+1}.

As in the base case (Type I), we cannot include w in the set of active convex vertices and $a_H(v)$ must be replaced by $a_V(c')$, because $a_H(v)$ becomes a reflex vertex when the rectangle is glued. Since the edge $e_H(v)$ is now the new horizontal cut, that cut

does not intersect any previous cut (by definition of the rectilinear partition). The new vertical cut, denoted by l_I, is directed to the exterior of P_r, and affects only the rectangle we glued. Therefore, it cannot intersect previous cuts either. The extreme piece is replaced by a chain of three pieces, and the new extreme piece is now of Type I. As before, the other extreme piece of $\Pi_{HV}(P_r)$ is kept as extreme piece of $\Pi_{HV}(P_{r+1})$. This concludes our proof. □

3.3.1 Implementation

To develop the generator for thin grid ogons, we adapted the original functions GenerateGridOgon and GetFSN, to take into account the structural properties derived from the proof of Proposition 3. The new version of GenerateGridOgon is presented as Algorithm 14.

Now, we assign a type to each convex vertex in C, which is the type of the extreme piece it belongs to. In addition, in case Type I, we associate the reflex vertex that yielded the piece to the two convex vertices. This allows us to identify the types easily, as well as the delimiting r-cuts.

The function InitTypes (line 14) performs the initialization of C and of the types in the first iteration and UpdateTypes (line 27) updates them in the following iterations.

The first iteration is handled separately because it is a bit different from the other ones, as the base case in the proof of Proposition 3. When $r = 0$, the grid ogon is the unit square. The computation of the free neighborhood is trivial. For that reason, we defined a specific function GetFSNBaseCase, presented below as Algorithm 15.

For the remaining iterations, we adapted the computation of $FSN(v)$ to make use of the type of the extreme piece that contains v and of the delimiting cuts, following the proof of Proposition 3. The new version is shown as Algorithm 16.

Algorithm 14 Customized version of GENERATEGRIDOGON to create thin grid ogons.

1: **procedure** GENERATEGRIDOGON(n)
2: INITSTRUCTURES();
3: $r \leftarrow (n-4)/2$;
4:
5: **if** $r \neq 0$ **then**
6: $v\_id \leftarrow$ GETRANDOMVERTEX(C);
7: $\{fsn, \triangle_H, \triangle_V\} \leftarrow$ GETFSNBASECASE(v_id);
8: $(c_x, c_y) \leftarrow$ GETRANDOMCELL($fsn, \triangle_H, \triangle_V$);
9:
10: $(c'_x, c'_y) \leftarrow$ INFLATE((c_x, c_y));
11: $a_H v\_id \leftarrow$ GETADJACENT(v_id, H);
12: $v_y \leftarrow$ GETCOORDINATE(v_id, y); $e_H v\_x\text{-}interval \leftarrow$ GETINTERVALEDGE(v_y, H);
13: $\{c\_id, a_V c\_id\} \leftarrow$ PASTE($v\_id, a_H v\_id, (c'_x, c'_y)$);
14: INITTYPES($v\_id, a_H v\_id, c\_id, a_V c\_id, (c'_x, c'_y), e_H v\_x\text{-}interval$);
15:
16: $dim \leftarrow dim + 1$;
17: $r \leftarrow r - 1$;
18: **end if**
19: **while** $r \neq 0$ **do**
20: $v\_id \leftarrow$ GETRANDOMVERTEX(C);
21: $\{fsn, \triangle_H, \triangle_V\} \leftarrow$ GETFSN(v_id);
22: $(c_x, c_y) \leftarrow$ GETRANDOMCELL($fsn, \triangle_H, \triangle_V$);
23:
24: $(c'_x, c'_y) \leftarrow$ INFLATE((c_x, c_y));
25: $a_H v\_id \leftarrow$ GETADJACENT(v_id, H);
26: $\{c\_id, a_V c\_id\} \leftarrow$ PASTE($v\_id, a_H v\_id, (c'_x, c'_y)$);
27: UPDATETYPES($v\_id, a_H v\_id, c\_id, a_V c\_id$);
28:
29: $dim \leftarrow dim + 1$;
30: $r \leftarrow r - 1$;
31: **end while**
32: **end procedure**

Algorithm 15 GetFSN for the unit square (i.e., when $r = 0$).

```
1: function GetFSNBaseCase(v_id)
2:     v_x ← GetCoordinate(v_id, x); v_y ← GetCoordinate(v_id, y);
3:     △_H ← GetWay(v_id, H); △_V ← GetWay(v_id, V);
4:
5:     r_y-start ← v_y;
6:     if △_V < 0 then
7:         r_x-interval ← ]0, v_x[;
8:     else
9:         r_x-interval ← ]v_x, dim + 1[;
10:    end if
11:    r_y-finish ← v_y + △_H;
12:    fsn ← {r};
13:
14:    return {fsn, △_H, △_V}
15: end function
```

The function GetRectangles is the same as before (see Algorithm 4). The first two parameters define the starting point and the V-line that limits the visibility from v. We use these parameters to customize GetFSN, as follows.

If v belongs to an extreme piece of Type I then l_x defines l_I, as we observe in Figure 3.9(a).

In the original version of the Inflate-Paste algorithm, l_x is the x-coordinate of the first V-edge that intersects the horizontal extension of $e_H(v)$ and blocks the rectangular visibility (or a V-line of the grid boundary when there is no such V-edge). However, from the proof of Proposition 3, if v belongs to a Type I piece, we know that l_I is a vertical limit that never surpasses a V-edge of this kind, since l_I is always a value between v_x and $a_H(v)_x$.

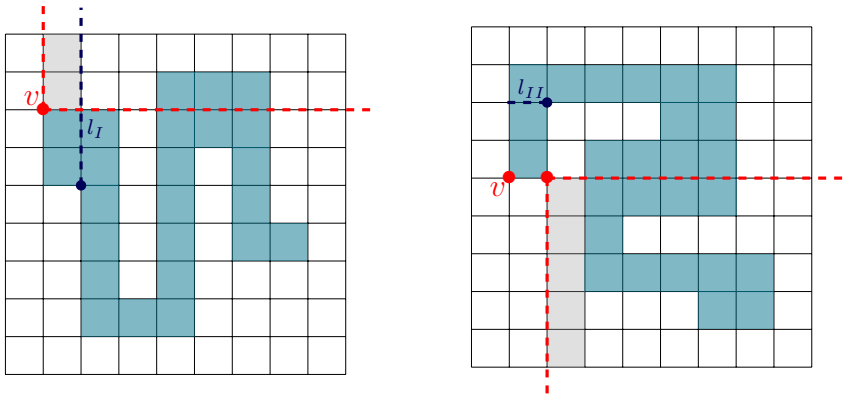
Algorithm 16 GetFSN for thin grid n-ogons, for $n \geq 6$.

function GetFSN(v_id)
  $v_x \leftarrow$ GetCoordinate(v_id, x); $v_y \leftarrow$ GetCoordinate(v_id, y);
  $\triangle_H \leftarrow$ GetWay(v_id, H); $\triangle_V \leftarrow$ GetWay(v_id, V);

  if GetType(v_id) $= I$ **then**
    $r_{l_I}\_id \leftarrow$ GetReflex$I$(v_id); $l_I \leftarrow$ GetCoordinate($r_{l_I}\_id, x$);
    $fsn \leftarrow$ GetRectangles($(v_x, v_y), l_I, \triangle_H, \triangle_V$);

  else //GetType(v_id) $= II$
    $a_H v\_id \leftarrow$ GetAdjacent(v_id, H);
    $a_H v_x \leftarrow$ GetCoordinate($a_H v\_id, x$); $a_H v_y \leftarrow$ GetCoordinate($a_H v\_id, y$);
    $l_x \leftarrow$ GetFSNx-Limit($(v_x, v_y), \triangle_H, \triangle_V$);
    $fsn' \leftarrow$ GetRectangles($(a_H v_x, a_H v_y), l_x, \triangle_H, \triangle_V$);
    $fsn \leftarrow$RefineFSNII($fsn', v\_id, \triangle_H$);
  end if

  return $\{fsn, \triangle_H, \triangle_V\}$
end function

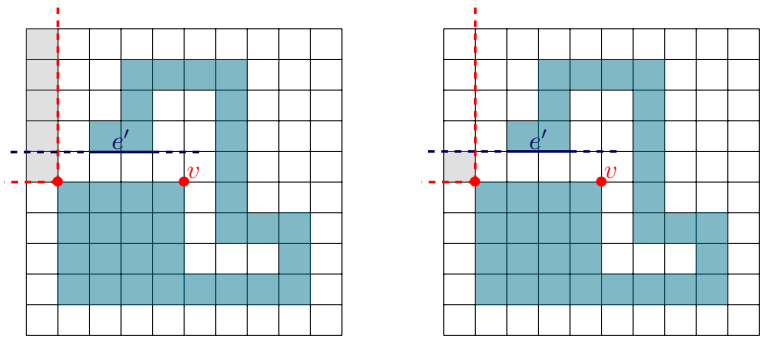


(a) (b)

Figure 3.9: (a) GetRectangles($(v_x, v_y), l_I, \triangle_H, \triangle_V$), for Type I. (b) GetRectangles($(a_H v_x, a_H v_y), l_x, \triangle_H, \triangle_V$), for Type II.

If v belongs to an extreme piece of Type II, the starting point is $(a_H(v)_x, a_H(v)_y)$ instead of (v_x, v_y). This means that we simply want all cells in $FSN(v)$ that are not between v_x and $a_H(v)_x$. In fact, from the proof of Proposition 3, we know that only cells surpassing $a_H(v)$ can be selected in this case.

This leads to the restriction illustrated in Figure 3.9(b). However, we may have to refine the region further if there are some other edges that obstruct visibility, as the H-edge e' in the example we show in Figure 3.10(a). For that purpose, we introduced a new function RefineFSNII, to check that and finally get the correct region (see Figure 3.10(b)).



(a) (b)

Figure 3.10: Case Type II: (a) $fsn' \leftarrow$ GetRectangles$((a_H v_x, a_H v_y), l_x, \triangle_H, \triangle_V)$. (b) $fsn \leftarrow$ RefineFSN$(fsn', v_id, \triangle_H)$.

As we mentioned above, the function InitTypes initializes C and the types in the first iteration and UpdateTypes updates them in the following iterations. These functions are presented in pseudocode as Algorithm 17 and Algorithm 18.

The auxiliary functions used in the algorithms are described in Table 3.1.

Algorithm 17 Setting set C and type initialization.

function InitTypes($v\_id, a_H v\_id, c\_id, a_V c\_id, (c'_x, c'_y), e_H v\_x\text{-}interval$)
  $a_V a_H v\_id \leftarrow$ GetAdjacent($a_H v\_id, V$);
  $C \leftarrow \{a_V a_H v\_id, c\_id\}$;

  if $c'_x \in e_H v\_x\text{-}interval$ **then**
    $C \leftarrow C \cup \{v\_id, a_H v\_id\}$;
    SetType($a_H v\_id, a_V a_H v\_id, I$); SetReflex$I$($a_H v\_id, a_V a_H v\_id, a_V c\_id$);
    SetType(v_id, c_id, II);
  else
    $a_V v\_id \leftarrow$ GetAdjacent(v_id, V);
    $C \leftarrow C \cup \{a_V c\_id, a_V v\_id\}$;
    SetType($c\_id, a_V c\_id, I$); SetReflex$I$($c\_id, a_V c\_id, a_H v\_id$);
    SetType($a_V v\_id, a_V a_H v\_id, II$);
  end if
end function

Algorithm 18 Updating set C and adding the type of the new extreme piece.

function UpdateTypes($v\_id, a_H v\_id, c\_id, a_V c\_id$)
  if GetType(v_id) $= I$ **then**
    $a_V v\_id \leftarrow$ GetAdjacent(v_id, V);
    $C \leftarrow (C \setminus \{a_V v\_id\}) \cup \{c\_id\}$;
    SetType(c_id, v_id, II);

  else //GetType(v_id) $= II$
    $C \leftarrow (C \setminus \{v\_id, a_H v\_id\}) \cup \{c\_id, a_V c\_id\}$;
    SetType($c\_id, a_V c\_id, I$); SetReflex$I$($c\_id, a_V c\_id, a_H v\_id$);
  end if
end function

Function	Description	Algorithms
SetType(p_id, q_id, τ)	Assigns the type τ to the vertices p_id and q_id.	17, 18, 19
SetReflex$I(p_id, q_id, r_{l_I}_id)$	Used for Type I cells to associate the reflex vertex $r_{l_I}\_id$ to the convex vertices p_id and q_id.	17, 18, 19
GetType(p_id)	Returns the type of the convex vertex p_id.	16, 18, 19
GetReflex$I(p_id)$	Returns the V-cut l_I associated to the convex vertex p_id.	16
RefineFSN$II(fsn', v_id, \triangle_H)$	Used if v is of Type II to refine fsn', ensuring that all points are rectangularly visible from v.	16

Table 3.1: Some auxiliary functions of the generator of thin grid ogons.

3.4 Spiral Grid n-Ogons

In [23], it was proved that the spiral grid n-ogons are a proper subclass of the thin grid n-ogons, and the following result.

Lemma 3. *Only spiral grid n-ogons can yield, by Inflate-Paste, spiral grid $(n+2)$-ogons.*

We exploit these properties to develop a generator for spiral grid n-ogons based on the one for the thin grid n-ogons.

All we need is to restrict the set C of active convex vertices, to keep all reflex vertices in one chain and all the convex vertices in another chain. For $r \geq 1$, there are only two active vertices in each iteration. By case analysis, we conclude that for a Type I extreme piece, the active convex vertex v is the one that has a reflex adjacent. This reflex vertex will become adjacent to the new reflex. If the piece is of Type II, we must select v as the convex vertex whose two neighbors are convex vertices. When we glue

the new rectangle, $a_H(c')$ is the new active convex vertex if the new piece is of Type II. Otherwise, if the new piece is of Type I, $a_V(c')$ becomes the new active convex vertex.

Hence, the thin grid ogons Inflate-Paste version can be adapted to create spiral grid ogons. We change the instructions that update set C in the InitTypes and UpdateTypes functions. In InitTypes (Algorithm 17), line 3 is replaced by the instruction $C \leftarrow \emptyset$. The modified UpdateTypes is described in Algorithm 19.

Algorithm 19 Customized version of UpdateTypes to create spiral grid ogons.

```
1: function UpdateTypes(v_id, a_H v_id, c_id, a_V c_id)
2:     if GetType(v_id) = I then
3:         SetType(c_id, v_id, II);
4:
5:     else  //GetType(v_id) = II
6:         C ← (C \ {v_id}) ∪ {a_V c_id};
7:         SetType(c_id, a_V c_id, I); SetReflexI(c_id, a_V c_id, a_H v_id);
8:     end if
9: end function
```

Chapter 4

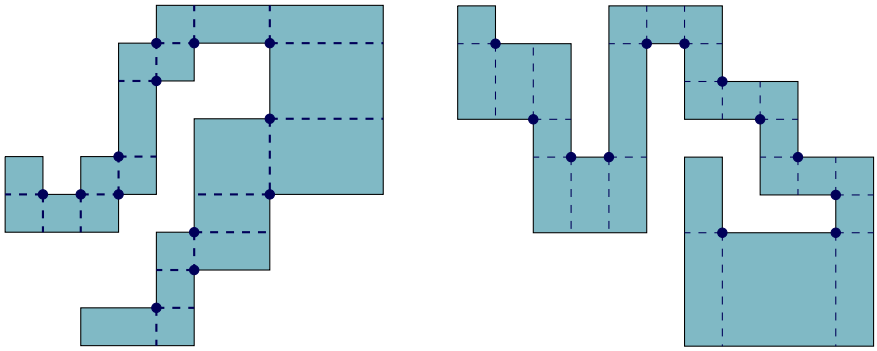
α-CAGP in Steiner Path Orthogonal Polygons

This chapter addresses the Chromatic Art Gallery Problem with edge-aligned vertex α-guards (α-CAGP), for $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$, focusing on Steiner path polygons. It is known that $\mathcal{X}_{\mathcal{G}_{2\pi}}(P) \leq 2$ for spiral polygons and $\mathcal{X}_{\mathcal{G}_{2\pi}}(P) \leq 3$ for staircase polygons [7]. We show that the chromatic α-guard number of any Steiner path orthogonal polygon P satisfies $\mathcal{X}_{\mathcal{G}_\alpha}(P) \leq 3$, for $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$, and this combinatorial bound is tight.

4.1 On the Structure of Steiner Path Ogons

A *path orthogonal polygon* is an orthogonal polygon P such that the dual graph of its rectilinear partition $\Pi_{HV}(P)$ is a path graph. When no rectilinear cut links two reflex vertices, which means that all horizontal and vertical cuts intersect the boundary at Steiner points (i.e., at extra points, other than vertices of P), we call such a polygon a *Steiner path orthogonal polygon* (see Figure 4.1).

All thin grid ogons are Steiner path polygons and we can obtain path polygons from thin grid ogons by shifting the grid lines. In addition, our generator GenerateOgon can produce both Steiner path orthogonal polygons and path orthogonal polygons that are not Steiner path orthogonal polygons by asking for polygons in general position or not in general position.



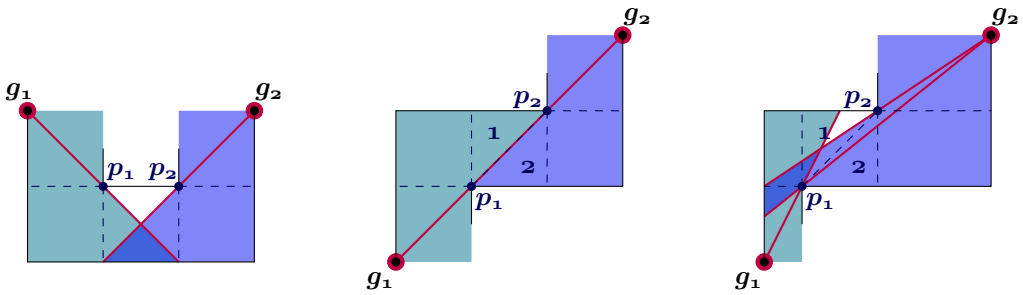
(a) (b)

Figure 4.1: (a) A path not Steiner path. (b) A Steiner path.

For a Steiner path polygon P with r reflex vertices, $\Pi_{HV}(P) = 2r + 1$. In our study of α-CAGP, we consider a decomposition of P into L-shaped pieces, where each piece consists of the three r-pieces that are incident to a given reflex vertex v of P. We call it a *turn-piece* (or the *turn-piece of* v).

In addition to this decomposition, we consider a refinement of $\Pi_{HV}(P)$ that turned out to be very useful also. The refinement is obtained by triangulating the r-pieces of $\Pi_{HV}(P)$ that contain two reflex vertices, as opposite corners. The r-piece is split into two triangles by adding the diagonal that links those vertices. We call each half a $\triangle$-piece. Using this partition, we can decide whether an r-piece could be covered *in cooperation* or whether the piece requires a guard that can see all its points.

We can restrict the analysis to r-pieces shared by two adjacent turn pieces. Let p_1 and p_2 be the two reflex vertices that define the two turn pieces. Up to symmetry, the relevant cases are the ones we sketch in Figure 4.2.



(a) (b) (c)

Figure 4.2: Checking whether g_1 and g_2 can jointly guard the r-piece that contains p_1 and p_2.

When p_1 and p_2 define an edge of the polygon (that is, the two turn-pieces define an U-shaped piece), as in Figure 4.2(a), two guards located at g_1 and g_2 cannot assist each other to cover the r-piece that contains the edge $\overline{p_1p_2}$. Actually, there would be points of P in a neighborhood of that edge that cannot be covered unless the guard sees all the points in the r-piece. So, the r-piece cannot be guarded in cooperation.

When p_1 and p_2 are in different chains (that is, the two turn-pieces define a staircase), as in Figures 4.2(b)–(c), g_1 and g_2 can assist each other to fully cover the r-piece with opposite corners p_1 and p_2 if and only if each of the two $\triangle$-pieces is completely covered by one of them (i.e., g_1 covers the $\triangle$-piece with label 1 and g_2 covers the $\triangle$-piece with label 2).

Indeed, as we see in Figure 4.2(c), if g_1 does not cover the $\triangle$-piece labelled 1, then, even if g_2 can cover some part of that piece, there would be some points near p_2 that were not guarded. The same holds in the case where g_2 does not cover all points of the $\triangle$-piece labelled 2, now for p_1 instead of p_2.

For solving α-CAGP in Steiner path polygons, we can exploit this property further. Let us assume that we are checking whether a given polygon P admits a 2-colorable guard set. Now, suppose that the staircase piece is part of a larger staircase, which includes vertex g that together with g_2 can see the piece defined by p_1 and p_2, as depicted in Figure 4.3.

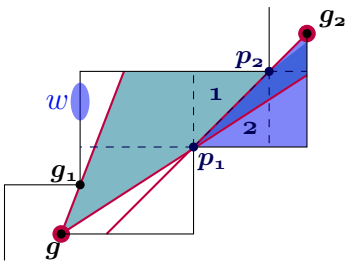


Figure 4.3: A fragment requiring 3 colors if we use g and g_2 to guard the r-piece given by p_1 and p_2.

Clearly, there are points of P in the neighborhood of $e_V(g_1)$ that cannot be guarded either by g_2 or g, for instance in the shaded region labeled with w. Since the two guards g and g_2, as well as any guard that can cover w see p_1, then there is a conflict between at least three guards. So, the guard set is not 2-colorable (as it requires at least three colors).

In the next sections, we will show that there always exist a 3-colorable α-guard set for any Steiner path orthogonal polygon and that there are instances that require three colors. The previous observations allow us to simplify the analysis of the problem.

4.2 Minimum Number of Colors for Steiner Path Ogons

We looked for combinatorial lower bounds for the chromatic guard number $\mathcal{X}_{\mathcal{G}_\alpha}(P)$, that can hold for every P, assuming edge-aligned vertex α-guards with $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$.

Although our first impression was that the problem could be simple for $\alpha = \frac{\pi}{2}$, in the end it turned out to be more difficult than the case when $\alpha = 2\pi$. Indeed, our initial conjecture that two colors could be sufficient if $\alpha = \frac{\pi}{2}$ was not true. We concluded that a Steiner path orthogonal polygon can have a long staircase requiring two colors and may require a guard in a vertex that sees all the vertices along the "diagonal" of another long staircase, leading to a 3-coloring. This cannot be avoided, even when we restrict to $\frac{\pi}{2}$-guards. The instances we constructed by hand for proving the lower bounds are of that kind and symmetric (e.g., the instance depicted in Figure 4.4).

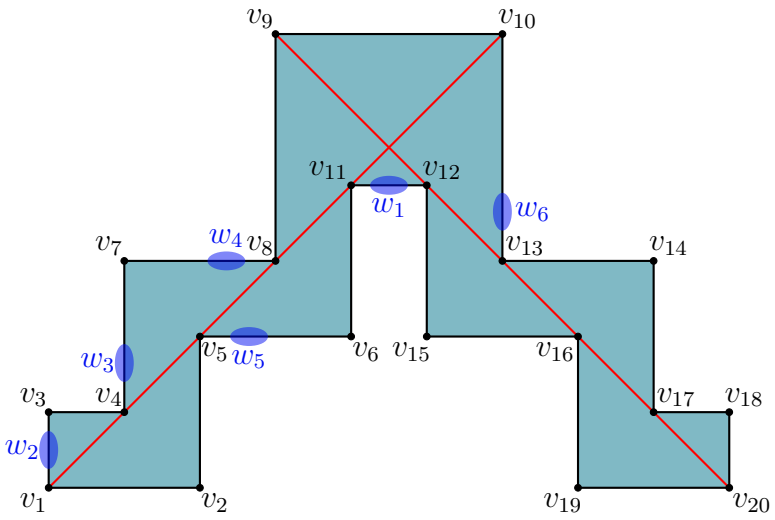


Figure 4.4: A Steiner path ogon that has no 2-colorable vertex 2π-guard set.

In the proof of Proposition 4, stated below, we show that this instance has no 2-colorable vertex 2π-guard set. The shaded ellipses indicate regions that will be useful to prove the existence of conflicts. We refer to these useful regions as *witness regions* (witnesses or critical regions).

Proposition 4. *For $n \geq 20$, there are Steiner path n-ogons P such that $\mathcal{X}_{\mathcal{G}_{2\pi}}(P) \geq 3$, for vertex 2π-guards.*

Proof. First we prove that the instance given in Figure 4.4 requires a 3-coloring.

Clearly, to cover the witness region w_1 (on the top of the polygon), we need a 2π-guard at either v_9, v_{10}, v_{11} or v_{12} and each of these vertices sees all the reflex vertices of P that belong to one the two staircases (i.e., that lie on one of the two oblique lines in • color).

Without loss of generality, let us focus on the staircase that contains v_1. The witness region w_2 requires a 2π-guard at either v_1, v_2, v_3 or v_4.

If the guard is located at either v_1 or v_3, we need another guard to cover the witness region w_3, because neither v_1 nor v_3 see w_3. The new guard must be located at either v_2, v_4, v_5, v_7 or v_8. All of these vertices, as well as v_1 and v_3, see the vertex v_4. Thus, if we locate a guard at v_1 or v_3, we need at least two colors for any guard set for that staircase.

Now, let us see what happens if we locate a guard at v_2 to cover w_2. Since v_2 cannot see the witness region w_4, we need another guard at either v_4, v_5, v_6, v_7 or v_8. All these vertices, as well as v_2, see the vertex v_5. Therefore, we need at least two colors for any guard set for that staircase if v_2 is used.

In both cases, if we place a 2π-guard at either v_{10} or v_{11} to cover the witness region w_1, then another conflict arises at v_4 or v_5, and hence we need three colors (at least).

Instead of v_{10} and v_{11}, we could select either v_9 or v_{12} to locate a guard to cover the witness w_1. Nevertheless, since the polygon is symmetric, similar conflicts will arise in the other staircase (with v_3, v_2 and v_1 replaced by v_{18}, v_{19} and v_{20} for the analysis).

Finally, we consider the case where the guard that covers w_2 is located at v_4. The analysis is slightly different. The witness w_5 requires another 2π-guard, which must be placed at either v_5, v_6, v_7, v_8 or v_{11}. All these vertices, as well as v_4, see v_8. So, we need at least two colors.

If we select v_{11} as the new guard, the witness region w_1 will be covered, but the witness w_6 will not. Hence, a third guard is needed to cover w_6 and it must be located at either v_9, v_{10}, v_{12}, v_{13} or v_{15}. All these vertices, as well as v_4 and v_{11}, see v_{10}. Therefore, we need at least three colors. Instead of v_{11}, we could select one of the vertices v_5, v_6, v_7 and v_8 to cover w_5. Then, we need one of the four vertices v_9, v_{10}, v_{11} or v_{12} to cover the witness w_1. But, v_{10} and v_{11} will raise a conflict at v_8, asking for a third color and, by symmetry, also v_9 and v_{12} will raise conflicts in the other staircase.

This concludes the proof that there is no 2-colorable vertex 2π-guard set for the Steiner path ogon given in Figure 4.4. From this instance, if we extend one of the staircases downwards, we get a family of Steiner Path n-ogons, with $n \geq 20$, whose chromatic guard number is greater than 2, vertex 2π-guards. □

Both for $\alpha = \pi$ and $\alpha = \frac{\pi}{2}$, the instance we give in Figure 4.4 admits a 2-colorable edge aligned α-guard set. Nevertheless, the instance depicted in Figure 4.5, which we call "the space invader", requires three colors in that case.

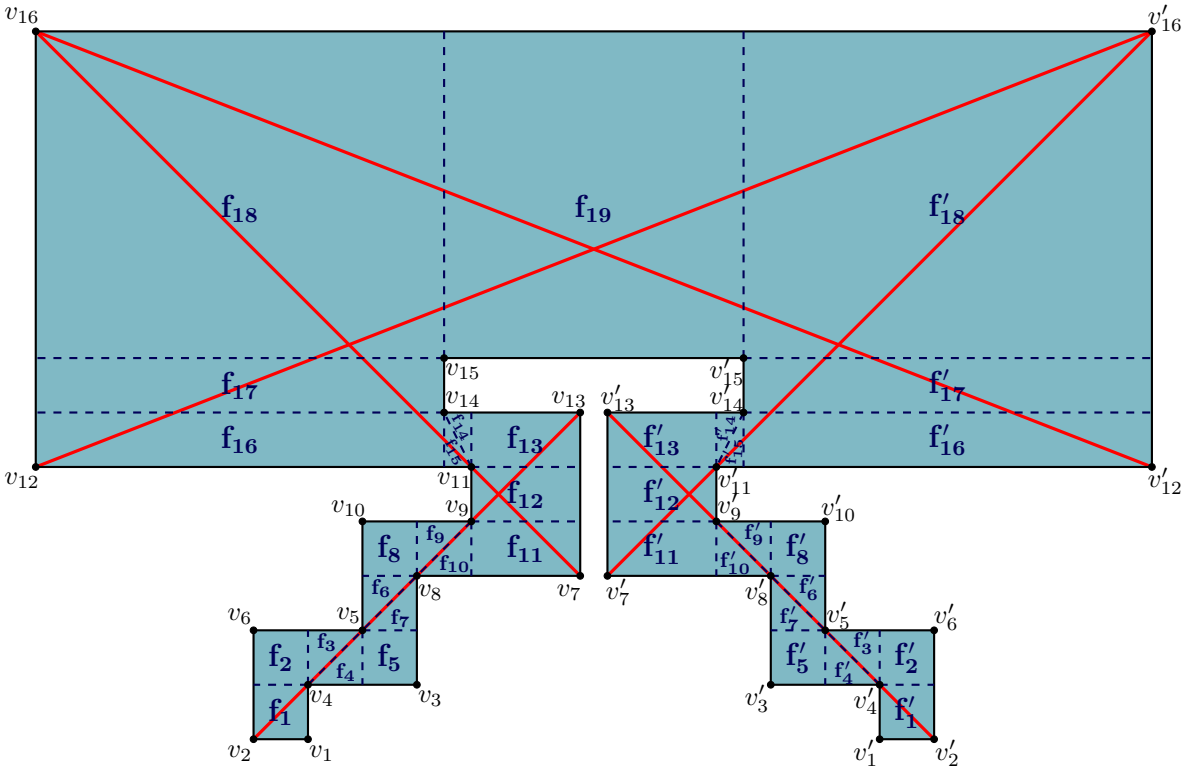


Figure 4.5: A Steiner path ogon that has no 2-colorable vertex α-guard set, for $\alpha = \pi, \frac{\pi}{2}$.

We constructed this instance by hand, after several attempts to design an algorithm for finding 2-colorable guard sets, for $\frac{\pi}{2}$-CAGP. The polygon is symmetric with respect to the bissector of the topmost H-edge and the labels we use in the right side of the picture, for the vertices and also for the r-pieces and $\triangle$-pieces, reflect their counterparts in left side.

Since the analysis of the problem was even more complex than the former one, we modeled this instance as a constraint satisfaction problem and used a constraint programming system to assist us in the proof. The mathematical model we present in this section is slightly different. It is based on *binary integer programming* and it is closed related to the generic model we implemented in our prototype solver for α-CAGP. This solver is described in the next chapter.

4.2.1 A BIP Model for Checking Whether $\mathcal{X}_{\mathcal{G}_\alpha}(P) \leq \kappa$

Let $\mathcal{G}$ be the candidate guard set and let $\mathcal{K}$ be the color set, with $|\mathcal{K}| = \kappa$. For every $g \in \mathcal{G}$ and $k \in \mathcal{K}$, we define a (binary) decision variable $x_{g,k}$ whose value is 1 if and only if g is selected and assigned color k. Otherwise, $x_{g,k} = 0$.

For $\alpha = \pi$ and $\alpha = \frac{\pi}{2}$, there are two possible ways of locating an edge-aligned α-guard at a reflex vertex v. We use v^H if it is aligned with the H-edge and v^V if it is aligned with the V-edge incident to v. Because of that we separate the set of convex vertices from the set of the reflex vertices, and denote them by $\mathcal{C}$ and $\mathcal{R}$, respectively. Then $\mathcal{G} = \mathcal{C} \cup \{v^H, v^V \mid v \in \mathcal{R}\}$.

The constraints of the model must guarantee that the selected guards cover the whole polygon, there is at most one guard at each vertex, and there are no color conflicts. The guarding problem is modeled as a set covering problem. For that purpose, we consider a decomposition Π of the polygon and require that each piece of Π can be covered by at least one guard in the guard set. Let F_f be the set of guards from $\mathcal{G}$ that can see a given piece $f \in \Pi$, and $\mathcal{F}$ be a set of all F_f, for $f \in \Pi$.

The following constraints require that the selected guard set covers all pieces of Π and that at most one guard is located at each vertex.

$$\sum_{g \in F} \sum_{k \in \mathcal{K}} x_{g,k} \geq 1, \qquad \text{for all } F \in \mathcal{F} \tag{4.1a}$$

$$\sum_{k \in \mathcal{K}} x_{v,k} \leq 1, \qquad \text{for all } v \in \mathcal{C} \tag{4.1b}$$

$$\sum_{k \in \mathcal{K}} (x_{v^H,k} + x_{v^V,k}) \leq 1, \qquad \text{for all } v \in \mathcal{R} \tag{4.1c}$$

We will now define constraints to discard color conflicts (incompatibilities). For that purpose, we identify a set of *critical* points or regions, and determine the subset I of $\mathcal{G}$ that can see each one. Let $\mathcal{I}$ be the set of all subsets I of (possibly) conflicting guards. The following constraint states that no two guards in such a set I can be assigned the same color.

$$\sum_{g \in I} x_{g,k} \leq 1, \qquad \text{for all } k \in \mathcal{K} \text{ and } I \in \mathcal{I} \tag{4.2a}$$

We will now see how we used this model to check that $\mathcal{X}_{\mathcal{G}_\alpha}(P) > 2$ holds for the space invader, for $\alpha = \frac{\pi}{2}, \pi$.

4.2.1.1 The Data for the "Space Invader" Model

As we discussed above, in Section 4.1, for checking whether a Steiner path polygon P admits a 2-colorable guard set, we can take Π as the refinement of $\Pi_{HV}(P)$ obtained by triangulating the r-pieces that have two reflex vertices as opposite corners. This is the partition we considered for the "space invader" polygon, as we can see in Figure 4.5. Since the polygon is symmetric, we use f_i to label the pieces on the left side and f'_i to label the corresponding piece on the right side, for $1 \leq i \leq 18$, and f_{19} to label the r-piece on the top. The set of α-guards that can see f_i is given by F_i and the same applies to f'_i and F'_i.

It is worthwhile noting that we are not looking for guard sets of minimum cardinality. Otherwise, in general, we need to take a partition that gives a more accurate model (for instance, the partition induced by the visibility regions of all guards [16, 17]).

In the same way, in order to show that $\mathcal{X}_{\mathcal{G}_\alpha}(P) > \kappa$, for some constant κ (for instance, $\kappa = 2$), we do not have to identify all possible color conflicts, we can prove that the constraints are already inconsistent for a proper subset. We followed this approach in the model defined for "space invader", and restricted the critical points to the vertices of the polygon.

Hence, the conflict set I_i refers to all guards that can see vertex v_i and we take advantage of symmetry, defining I'_i as the set of all guards that can see the vertex v'_i, for $1 \leq i \leq 16$.

Since the model is incomplete, if we find a solution using κ colors, $\mathcal{X}_{\mathcal{G}_\alpha}(P) \not> \kappa$ can be a false negative and we must check whether or not $\mathcal{G}_\alpha$ raises other conflicts that were not taken into account in the data.

Nevertheless, when we defined $\mathcal{I} = \{I_i, I'_i \mid 1 \leq i \leq 16\}$ and $\kappa = 2$, the model for the "space invader" instance was inconsistent, implying that $\mathcal{X}_{\mathcal{G}_\alpha}(P) > 2$, for $\alpha = \pi, \frac{\pi}{2}$.

The data is given in Figures B.1 and B.2, in Appendix B. We omitted the values of F'_i and I'_i because they can be recovered (by the reflection) from the values of F_i and I_i, for all i.

As for the previous case, we can extend one of the staircases to define a family of Steiner path n-ogons, for $n \geq 32$, with similar properties. This concludes the proof of Proposition 5.

Proposition 5. *For $n \geq 32$, there are Steiner path n-ogons P such that $\mathcal{X}_{\mathcal{G}_\alpha}(P) \geq 3$, for edge-aligned vertex α-guards with $\alpha = \pi$ and $\alpha = \frac{\pi}{2}$.*

4.3 Maximum Number of Colors for Steiner Path Ogons

In this section, we show that three colors are always sufficient for any Steiner path orthogonal polygon, that is, we prove the upper bound $\mathcal{X}_{\mathcal{G}_\alpha}(P) \leq 3$, for $\alpha = \frac{\pi}{2}, \pi, 2\pi$. The proof defines an algorithm, linear in n, that yields a 3-colorable guard set for the instance given as input.

Proposition 6. *Any Steiner path ogon P can be covered by edge-aligned vertex $\frac{\pi}{2}$-guards and using at most 3 colors.*

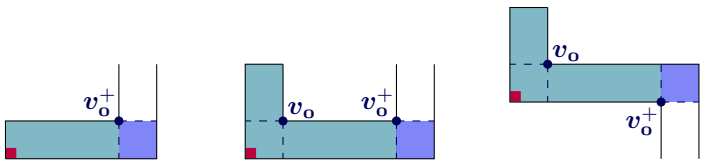
Proof. Let us consider a decomposition of P in turn-pieces, each one defined by some reflex vertex of P. For a given turn-piece, the turn-piece can have two possible orientations only (up to symmetry) and, therefore, in three consecutive turn-pieces we have four cases to check.

Let $r = 2m + s$, with $0 \leq s \leq 1$ and $m \geq 0$, and define an enumeration of the reflex vertices, starting from one endpoint of the path. For all $0 \leq k \leq m$, the label v_k is assigned to the reflex vertex $2k+s$ and v_k^+ to the following one (i.e., to the reflex vertex $2k + s + 1$).

We proceed by induction on k, preserving the following invariant.

- P can be totally covered by vertex $\frac{\pi}{2}$-guards, with a 3-coloring, until the first two r-pieces incident to v_k^+, in the enumeration order;
- There is at most two vertex $\frac{\pi}{2}$-guards seeing the r-piece that is incident to v_k^+ but not to the two adjacent reflex vertices (we call it the exclusive piece of v_k^+);
- There is at most one vertex $\frac{\pi}{2}$-guard seeing regions that are after the exclusive piece of v_k^+.

Base case: Figure 4.6 shows the location of guards for $k = 0$, for all cases, up to symmetry.

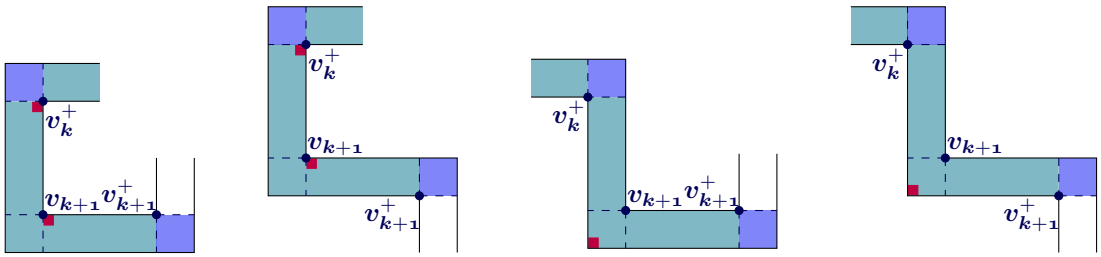


(a) (b)

Figure 4.6: The $\frac{\pi}{2}$-guards placement (● color) to cover the regions (● color) with the first s reflex vertices of P, including the first two v_0^+'s r-pieces. (a) The case $s = 0$. (b) The cases $s = 1$.

Induction step: Now, let us assume that for step k all the three properties referred above are kept, and check that the same can happen for step $k + 1$. Figure 4.7 depicts the four cases for the turn at v_{k+1} and the proposed location for the guards for them

(we always ignore symmetries and the true dimensions). By construction, there is at most one guard at each vertex.



(a) (b) (c) (d)

Figure 4.7: The $\frac{\pi}{2}$-guards placement (● color) to cover all the v_{k+1}'s r-pieces and the exclusive piece of v_{k+1}^+. Such regions are colored by ● and ●. In all cases, locally, the $\frac{\pi}{2}$-guards are 2-colored and there is at most one of them seeing the exclusive piece of v_{k+1}^+ (which has ● color), possibly surpassing it. Moreover, there is exactly one guard seeing something on the exclusive piece of v_k^+ (which has also ● color) but that cannot see anything before that.

By the induction hypothesis, all the r-pieces from the first $2k + s$ reflex vertices of P are covered and the exclusive piece of v_k^+ too, and 3 colors are sufficient. Furthermore, there is at most two guards seeing the exclusive piece of v_k^+ and at most one guard, g, that could see regions after this r-piece.

In Figure 4.7, we can check that, with the additional guards, P is totally covered until the first two r-pieces incident at v_{k+1}^+. At this point, there is no additional guard seeing something before the exclusive piece of v_k^+, only one seeing this r-piece, and these new guards are 2-colorable so that we can use 3 colors in total to include the possible g's conflicts.

From Figure 4.7, we see that only g and one of the new guards may see the exclusive piece of v_{k+1}^+.

Finally, we check that there is no more than one guard seeing regions after the exclusive piece of v_{k+1}^+. It is easy to see that this property holds in the cases (a) and (b). For the cases (c) and (d), the guard g may possibly have visibility rays that surpass v_{k+1}^+. However, by inspection of these two cases, we see that only one of the guards, either g or the new one, can see more than the exclusive piece of v_{k+1}^+. Hence, only one guard can see regions after that exclusive piece, implying that the third property also holds. □

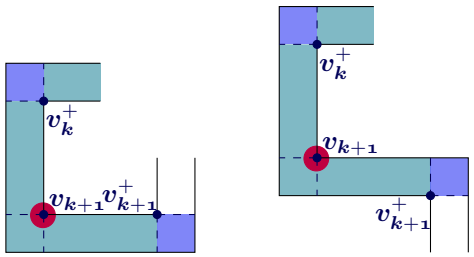
The proof of Proposition 6 can be easily adapted for a 3-coloring by π-guards and 2π-guards. Corollary 1 and Corollary 2 state these results.

Corollary 1. *Any Steiner path orthogonal polygon can be covered by vertex π-guards and using at most 3 colors.*

Proof. Exactly the same, but every $\frac{\pi}{2}$-guard is replaced by a π-guard. □

Corollary 2. *Any Steiner path orthogonal polygon can be covered by vertex 2π-guards and using at most 3 colors.*

Proof. The proof is similar, with every guard replaced by a 2π-guard, except that for the cases (a) and (b) of the induction step (see Figure 4.7) the new location for the guards is as we show in Figure 4.8.



(a) (b)

Figure 4.8: The new location of 2π-guards, replacing case (a) and (b) of Figure 4.7.

□

These proofs give us $O(n)$ time algorithms for finding a 3-colorable guard set in each case.

As regards the cardinality of the computed guard set, at least $\lfloor \frac{n}{4} \rfloor$ guards are necessary for all the three ranges, because the best case occurs when we add exactly one guard for every k, with $0 \leq k \leq \lfloor \frac{r}{2} \rfloor$ (we recall that $\lfloor \frac{r}{2} \rfloor + 1 = \lfloor \frac{n}{4} \rfloor$).

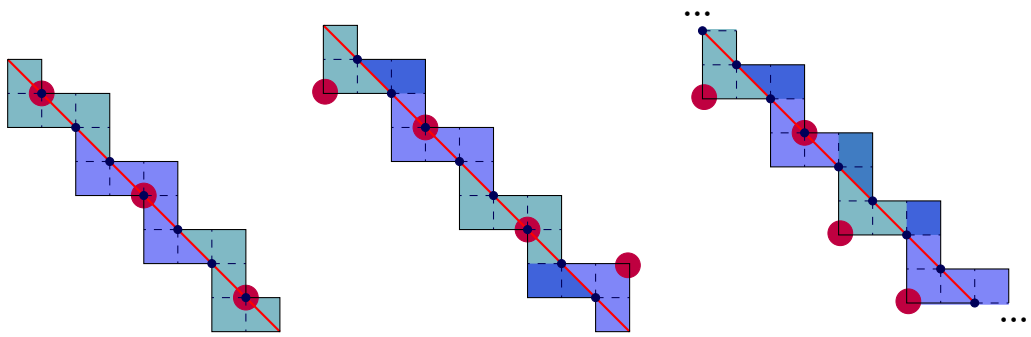
For the worst case, assuming $\frac{\pi}{2}$-guards or π-guards, we need $\frac{n}{2} - 1$ guards, since we may have to add two guards for each k, with $1 \leq k \leq \lfloor \frac{r}{2} \rfloor$ plus another for $k = 0$.

Thus, we have $2 \times \lfloor \frac{r}{2} \rfloor + 1 = 2 \times \lfloor \frac{n}{4} \rfloor - 1$ and

$$2 \times \lfloor \frac{n}{4} \rfloor - 1 = \begin{cases} \frac{n}{2} - 1 & \text{if } n \text{ is a multiple of 4} \\ \frac{n}{2} - 2 & \text{otherwise} \end{cases} .$$

Nevertheless, for 2π-guards, the upper bound is $\lfloor \frac{n}{4} \rfloor$, since we always add exactly one guard for each k, with $0 \leq k \leq \lfloor \frac{r}{2} \rfloor$, in all cases. Thus, assuming 2π-guards, we actually use $\lfloor \frac{n}{4} \rfloor$ guards to cover any Steiner path n-ogon.

We know that this number is tight for the *min-area grid orthogonal polygons* [2], which are Steiner path grid ogons that consist of a staircase formed by $2r + 1$ unit squares. For all $r \geq 9$, they require $\lfloor \frac{n}{4} \rfloor$ vertex 2π guards, for a 3-coloring. As we illustrate in Figure 4.9, for $r = 7$ and $r = 8$, we can have smaller guard sets.



(a) (b) (c)

Figure 4.9: (a) $r = 7$ ($n = 18$) and three 2π-guards are enough. (b) $r = 8$ ($n = 20$) and four 2π-guards are enough. (c) $r \geq 9$ ($n \geq 22$) and $\lfloor \frac{n}{4} \rfloor$ are required.

Let us assume that we may use at most three colors to cover min-area grid ogons and, above all, we want to minimize the number of vertex 2π-guards rather than the number of colors.

As we can see in Figure 4.9, for a min-area grid ogon P, a pair of reflex vertex 2π-guards can cover at most eleven consecutive r-pieces in $\Pi_{HV}(P)$. On the other hand, a pair of 2π-guards, one at a convex vertex and other at a convex or reflex vertex, can cover at most nine r-pieces.

It is noted that a 2π-guard at some reflex vertex has a vision ray traversing the staircase (color • in Figure 4.9). The min-area grid 18-ogon can be fully covered using only three

2π-guards at reflex vertices, but inevitably consuming three colors. The min-area grid 20-ogon cannot be fully covered by the same way, it costs at least four colors if we used more than two 2π-guards at reflex vertices. Four vertex 2π-guards is the optimal solution, but, once again, three colors have to be consumed. For min-area grid n-ogons with $n \geq 22$, we cannot use 2π-guards at two or more reflex vertices; otherwise we need more than three colors.

With this in mind, we now determine the minimum number of vertex 2π-guards needed for any min-area grid n-ogon with $n \geq 22$. Since only one reflex vertex 2π-guard can be used in total, and therefore it can cover as many pieces as a convex vertex 2π-guard, we can restrict to guards at convex vertices. Let us start to cover the region of an endpoint of the path (an extreme r-piece) and let us call the reflex vertex that induced that extreme r-piece as v.

The first 2π-guard we place sees at most four r-pieces. This maximum number is reached if the 2π-guard is at the convex vertex of the r-piece incident to v. Together, this 2π-guard and the next one can cover at most eight consecutive r-pieces (by putting the second on the convex vertex of the incident r-piece of the third reflex vertex). After the third 2π-guard placement, we have twelve consecutive r-pieces guarded. This placement is 2-colorable.

In summary, four r-pieces can be guarded per vertex 2π-guard, which means that $\lceil \frac{|\Pi_{HV}(P)|}{4} \rceil$
$= \lceil \frac{2r+1}{4} \rceil = \lceil \frac{n-3}{4} \rceil$ 2π-guards are needed. Note that

$$\lceil \frac{n-3}{4} \rceil = \begin{cases} \lceil \frac{4k-3}{4} \rceil = \lfloor \frac{n}{4} \rfloor & \text{if } n = 4k \\ \lceil \frac{4k+2-3}{4} \rceil = \lfloor \frac{n}{4} \rfloor & \text{otherwise} \end{cases} .$$

4.4 Related Work

In a recent paper [14], Fekete et al. present several results on the complexity of CAGP, including the following theorem:

Theorem 1. *For a simple polygon P with a finite candidate 2π-guards set $\mathcal{G}_{2\pi}$, it can be decided in polynomial time whether there is a 2-colorable 2π-guard set.*

The authors stated several properties about a 2-colorable 2π-guard set on simple polygons, that are important to formulate the algorithm. In particular, they showed, for a 2-colorable 2π-guard set, all vertices in the overlay of the 2π-guards visibility polygons lie on ∂P. Therefore, they may conclude that the dual graph of a triangulation defined with these vertices is a tree, in which every edge of a triangle can split P into two subpolygons, at most. They exploit this property to design a polynomial-time decision algorithm, based on Dynamic Programming.

This work does not deal with α-guards for $\alpha < 2\pi$, but we think that the algorithm works fine for α-CAGP also, with edge-aligned vertex guards. Therefore, we wonder whether we could have found more quickly a Steiner path orthogonal polygon that is not 2-colorable if we had run this algorithm for many instances in that family.

Chapter 5

Solving α-CAGP

Over the years, research in the area of AGP has led to combinatorial bounds, algorithms and hardness results for several variations of the problem.

In the previous chapter, we followed the first line of research, exploiting the geometric structure of the Steiner path orthogonal polygons to derive tight bounds for α-CAGP on that class. Although it seemed doable for the path orthogonal polygons as well, the analysis of the problem for that class turned out to be much more complex. For that reason, we decided to implement a prototype solver for α-CAGP, to be able to look for instances or check our conjectures more quickly, using a computer. In the end, we employed this solver to carry out an empirical study of α-CAGP on random path orthogonal polygons.

In this chapter, we address the design and implementation of our solver, and present the results from our empirical study. It is worthwhile noting the algorithms derived from the proofs of Proposition 6 and of its corollaries, in the previous chapter, compute a 3-colorable α-guard set for every Steiner path orthogonal polygon P, but cannot be used to find an optimal solution to α-CAGP on P.

In the last decade, several works in the area of AGP (and, more recently, CAGP [34, 33]) focused on the design of algorithms for computing optimal or good solutions for (possibly large) problem instances, as well as on their implementation and empirical evaluation. For a thorough survey on experimental work in this area of AGP, see [27]. Our solver is closely related to the work by Zambon et al. [34].

In a preprocessing phase, the problem is discretized, by making use of packages from CGAL. Then, it is modeled as a 0-1 ILP problem (a.k.a. Binary Integer Programming (BIP)) and solved using mathematical software for optimization (we used GNU Linear Programming Kit (GLPK)[1] instead of CPLEX).

We will start by describing the mathematical model and then explain the discretization procedure.

5.1 The Mathematical Model

As for the "space invader" instance, presented in the previous chapter (in Section 4.2), the discretization of α-CAGP corresponds to a problem α-CAGP$(\mathcal{W}, \mathcal{I}, \mathcal{G}, \mathcal{K}, \mathcal{C}, \mathcal{R})$, where $\mathcal{W}$ is a set of *witnesses* for visibility coverage, $\mathcal{I}$ is a set of *incompatibilities* (conflicting guard sets), $\mathcal{G}$ is the candidate guard set, $\mathcal{K}$ is the set of colors that can be used (we assume that $\mathcal{K} = \{1, 2, \ldots, \kappa\}$, where $\kappa = |\mathcal{K}|$), and $\mathcal{C}$ and $\mathcal{R}$ are the sets of convex vertices and reflex vertices of the polygon.

Each $W \in \mathcal{W}$ represents the set of guards that see a particular point/region in the polygon. It is assumed that once all such points/regions are guarded, the polygon is completely guarded. Therefore, the guarding problem corresponds to the instance of the set covering problem determined by $\mathcal{W}$. The relevant points/regions and the corresponding set $\mathcal{W}$ are computed in the preprocessing phase, as well as the set $\mathcal{I}$.

The candidate guard set $\mathcal{G}$ is given by $\mathcal{G} = \mathcal{C} \cup \{v^H, v^V \mid v \in \mathcal{R}\}$, for $\alpha = \pi$ and $\alpha = \frac{\pi}{2}$, and by $\mathcal{G} = \mathcal{C} \cup \mathcal{R}$, if $\alpha = 2\pi$. As in the "space invader" model, we use v^H if the guard at v is aligned with the H-edge incident to v and v^V if the guard is aligned with the V-edge. We restrict to orthogonal polygons.

Our model is based on the one given in [34, 33] for CAGP. To reduce its size and the computation time, we restrict $\mathcal{G}$ to the set of guards that can see some witness, that is, we reduce $\mathcal{G}$ to $\mathcal{G} \cap (\bigcup_{W \in \mathcal{W}} W)$. Then, the decision variables are defined as follows:

- For every $g \in \mathcal{G}$ and $k \in \mathcal{K}$, we define a (binary) decision variable $x_{g,k}$ whose value is 1 if g is selected and assigned color k. Otherwise, $x_{g,k} = 0$.

[1] https://www.gnu.org/software/glpk/

- For every $k \in \mathcal{K}$, we define a (binary) decision variable c_k whose value is 1 if the color k is used. Otherwise, $c_k = 0$.

- The variable z represents the value of the objective function, which is given by the sum of the values of the variables c_k, for $k \in \mathcal{K}$.

We look for solutions that minimize the value of z, under the constraints stated below.

$$z = \sum_{k \in \mathcal{K}} c_k \tag{5.1a}$$

$$\sum_{g \in W} \sum_{k \in \mathcal{K}} x_{g,k} \geq 1, \qquad \forall W \in \mathcal{W} \tag{5.1b}$$

$$\sum_{g \in I} x_{g,k} - c_k \leq 0, \qquad \forall I \in \mathcal{I},\ \forall k \in \mathcal{K} \tag{5.1c}$$

$$\sum_{g \in \mathcal{G}} x_{g,k} - |\mathcal{G}| c_k \leq 0, \qquad \forall k \in \mathcal{K} \tag{5.1d}$$

$$\sum_{k \in \mathcal{K}} x_{v,k} \leq 1, \qquad \forall v \in \mathcal{C} \tag{5.1e}$$

$$\sum_{k \in \mathcal{K}} (x_{v^H,k} + x_{v^V,k}) \leq 1, \qquad \forall v \in \mathcal{R} \tag{5.1f}$$

$$c_k - c_{k+1} \geq 0, \qquad \forall k \in \mathcal{K} \setminus \{\kappa\},\ \text{where } \kappa = |\mathcal{K}| \tag{5.1g}$$

$$\sum_{g \in \mathcal{G}} x_{g,k} - \sum_{g \in \mathcal{G}} x_{g,k+1} \geq 0, \qquad \forall k \in \mathcal{K} \setminus \{\kappa\} \tag{5.1h}$$

$$\sum_{g \in \mathcal{G}} x_{g,k} - c_k \geq 0, \qquad \forall k \in \mathcal{K} \tag{5.1i}$$

These constraints express the α-CAGP for $\alpha = \pi, \frac{\pi}{2}$. For case $\alpha = 2\pi$, the model is similar, except that the constraints 5.1e–5.1f are replaced by $\sum_{k \in \mathcal{K}} x_{v,k} \leq 1$, for all $v \in \mathcal{C} \cup \mathcal{R}$. These constraints require that there is at most one guard at each vertex of P. The remaining constraints establish that:

- 5.1b: the set of witnesses (for visibility coverage) is covered, that is, for all $W \in \mathcal{W}$, there is at least a colored guard selected from W;

- 5.1c: there are no color conflicts, that is at most one guard that belongs to a conflict set $I \in \mathcal{I}$ can get color k, for all $k \in \mathcal{K}$;

- 5.1d: if the color k is assigned to some guard then $c_k = 1$;
- 5.1g: the color $k+1$ will not be used if k is not used;
- 5.1h: the number of guards assigned color k is greater than or equal to the number with color $k+1$;
- 5.1i: if no guard g is assigned color k then $c_k = 0$.

The constraints 5.1g−5.1h are symmetry breaking constraints. They preserve the value of the optimal solution (that is, z) but reduce the search space by breaking some trivial symmetries. Considering that we look for solutions that minimize z, the constraint 5.1i is redundant. Nevertheless, it adds some information that can be useful to the solver.

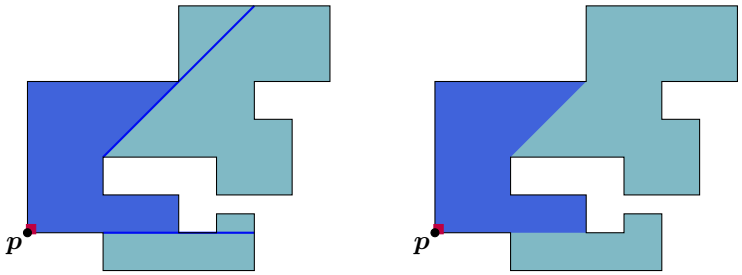
In order to define $\mathcal{I}$, in the preprocessing phase, we determine *the conflict graph* $(\mathcal{G}, \mathcal{P})$: two vertices g_1 and g_2 are linked by an edge in $\mathcal{P}$ if there is a point in the polygon that both g_1 and g_2 can see. In addition to $\mathcal{P}$, we consider a partition of the conflict graph into disjoint cliques and add these cliques to $\mathcal{I}$. Although the corresponding constraints add redundant cuts to the model, they can be useful to improve the performance of the ILP solver. As a side effect, we can discard the constraint 5.1d, since we will include in $\mathcal{I}$ also the cliques that consists of a single guard g. So, when we require that $x_{g,k} \leq c_k$, for all $k \in \mathcal{K}$, for $I = \{g\}$, we guarantee that even in that case the guard cannot be colored with color k unless $c_k = 1$.

5.2 The Discretization Procedure

As we mentioned above, the problem is discretized by running a preprocessing phase to determine the set of witnesses $\mathcal{W}$ and the set of incompatibilities $\mathcal{I}$.

For that purpose, it is necessary to compute the region that is visible from each guard. As we see in Figure 5.1, the visibility region may be a *degenerate* polygon (a.k.a. a *non-regularized* polygon), containing some needles.

These needles could be discarded in a model for AGP, but cannot be discarded in CAGP, where we have to ensure that there would be no color conflicts.



(a) (b)

Figure 5.1: (a) The non-regularized visibility polygon of vertex $\frac{\pi}{2}$-guard at p. (b) The regularized visibility polygon of vertex $\frac{\pi}{2}$-guard at p.

A polygon P is a *regularized* or *non-degenerate* polygon if its boundary does not have vertices nor edges lying on each other, i.e., $\partial(P \setminus \partial P) \cup (P \setminus \partial P) = closure(P \setminus \partial P)$, in the topological sense. Otherwise, P is called *non-regularized* or *degenerate*. Therefore, in α-CAGP, we have to be able to handle non-regularized visibility polygons to account for conflicts, whereas we can restrict the analysis of visibility coverage to regularized visibility polygons.

5.2.1 The Discretization of the Guarding Problem

In order to discretize the guarding problem, we determine a decomposition Π of the polygon P into a finite set of regions, such that for each $f \in \Pi$ and for each α-guard g in $\mathcal{G}$, either g sees no point in the interior of f or g sees all points in f. Such a region is called Atomic Visibility Polygon (AVP).

A natural decomposition satisfying this property is the one induced by the arrangement of line segments defined by all edges in ∂P and all edges of the regularized visibility polygons $\mathcal{V}(g, C_\alpha^g)$, for $(g, C_\alpha^g) \in \mathcal{G}_\alpha$. This arrangement splits P into closed regions (faces of the arrangement), each one being an AVP [16, 17, 6]. Nevertheless, the number of regions (pieces) can be too large. On the other hand, some of these regions may be redundant (or dominated) in the following sense.

Let $\mathcal{S}(f)$ denote the set of α-guards in $\mathcal{G}_\alpha$ that can see f. Define a strict partial order $\prec$ on the set of atomic visibility regions by $f_1 \prec f_2$ if and only if f_1 and f_2 are adjacent in Π and $\mathcal{S}(f_1) \subset \mathcal{S}(f_2)$, for all $f_1, f_2 \in \Pi$.

Clearly, we can restrict the witnesses for visibility coverage to the *minimal* elements in that strict partial order. Once the minimal elements are guarded, so is the polygon P.

The minimal elements are called the *shadow AVPs*, in the literature [34, 33]. The maximal elements are called *light AVPs*. Since every AVP f is convex, we can easily represent it by a single point w_f from its interior (e.g., the centroid of a triangle defined by three points in f that are not collinear). Let W_f be the set of α-guards that see the *witness point* w_f. We will define $\mathcal{W}$ as the set of all sets W_f, with f restricted to the shadow AVPs.

5.2.1.1 Implementation

For the implementation, we make use of some packages from CGAL. First, for each candidate guard, we compute the visibility polygons, both the regularized and non-regularized ones. This is done using the library <CGAL/Simple_polygon_visibility_2.h>, from *2D Visibility Computation* package.

For $\frac{\pi}{2}$-CAGP and π-CAGP, we have to take into account that, at each reflex vertex, there are two different candidate guards with different visibility cones. Currently, the package *2D Visibility Computation* provides results for 2π-guards only. However, it is easy to see that the regions we need for $\alpha \in \{\frac{\pi}{2}, \pi\}$ can be obtained by performing an intersection of the visibility polygon of the 2π-guard with suitable rectangles (which define the corresponding visibility cone). In Figure 5.2 we show some examples.

To perform these intersections and also the ones we need to identify conflicting guards, we use the library <CGAL/Nef_polyhedron_2.h>, from the *2D Boolean Operations on Nef Polygons* package.

Once we have computed the visibility polygons, we look for the shadow AVPs and their witness points. We make use of the package *2D Arrangements* to construct the arrangement defined by the edges of ∂P and the edges of the regularized visibility polygons. We find the shadow AVPs in a subsequent step.

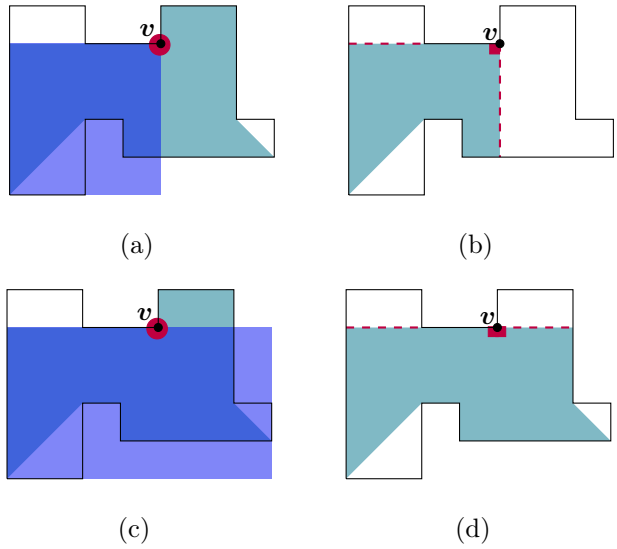


Figure 5.2: Finding visibility regions of α-guards by clipping the region visible to a 2π-guard, for $\alpha \in \{\frac{\pi}{2}, \pi\}$.

The arrangement is represented by Doubly-Connected Edge List (DCEL), which is a geometric data struture quite convenient for exploring the decomposition. It keeps information about the faces, edges, and vertices, incidence relations and adjacency relations. Each edge is represented by two twin edges (with opposite orientations and called *halfedges*). The faces are CCW oriented, meaning that the face is to the left of every halfedge that delimits it. The twin edges are halfedges of the adjacent faces.

The *2D Arrangements* package provides the library <CGAL/Arr_extended_dcel.h> that supports the DCEL representation and allows to store extra data on faces, halfedges and vertices. In our case, we need to save extra data on halfedges. The arrangement of AVPs is built by adding one segment (i.e., two halfedges) per iteration. We use the <CGAL/Arr_observer.h> library to update the information of the halfedges, when each halfedge pair is inserted to the arrangement.

We represent all the regularized visibility polygons and the polygon itself by DCEL data structures (i.e., a DCEL for each one, having the unbounded face, besides the bounded face that defines the polygon). Then, we assign a label to each halfedge of such polygons.

For every regularized visibility polygon, we assign label l to all the internal halfedges (the *CCW halfedges*), to represent the (enlighted) region which the guard can see. We

assign label s to all the external halfedges (the *clockwise (CW) halfedges*), to represent the (shaded) region which the guard cannot see.

Suppose that we overlay two visibility polygons, already labeled with l/s, and the overlay induces to three faces. Each of these faces inherits the labels that are in the edges of the visibility polygons. The face that both guards can cover has l in all its CCW halfedges and s in all the twin halfedges. Note that this makes sense, the face is completely enlighted by those guards but the outer faces are not. Each of the other two faces have l (s) in the interior (exterior) halfedges, which were inherited from the halfedges of the visibility polygon where the face belonged to.

With this in mind, halfedges of edges on regularized visibility polygons that are from edges on the boundary of the polygon instance should be assigned label n ("neutral") instead of l/s. The reason is that, actually, these halfedges are not delimited by guards visibility, they are delimited by walls of the polygon instance instead. Therefore, we set label n to all the halfedges of the polygon instance and, during the construction of the arrangement, we make sure that n has dominance to l/s values when halfedges collide. In other words, when two halfeges with same orientation and slope collide and one has label n and the other has label l or s, the resulting halfedge is always labeled with n.

Once the overlay of all visibility polygons and polygon instance is finished, we search for the shadow AVPs. This is done by traversing every face, one by one, and finding the ones that do not have l assignments in their interior halfedges. These faces are the shadow AVPs [33].

Note that guards that see a shadow face can see faces that are in the visibility polygons that have delimited this shadow face, by some of their edges. Furthermore, any face that shares an edge with the shadow face f is covered by all guards that cover f (unless the edge is from two visibility polygons that only intersect in the boundary). The face f is free of l assignments in its interior because, otherwise, there is an adjacent face f' with s in the halfedge that is twin to the interior halfedge of f with label l, implying that there is a guard $g \in \mathcal{S}(f)$ such that $g \notin \mathcal{S}(f')$ but this is absurd, since f is a minimal of f'. By a similar reasoning, all internal halfedges in a light AVP do not have s assignments.

In Figure 5.3, we provide an example of the AVPs obtained in a polygon, using vertex 2π-guards.

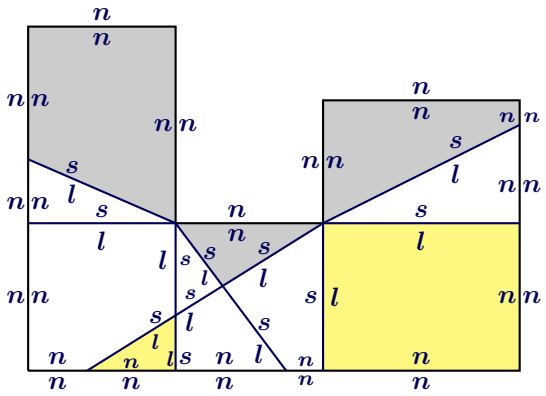
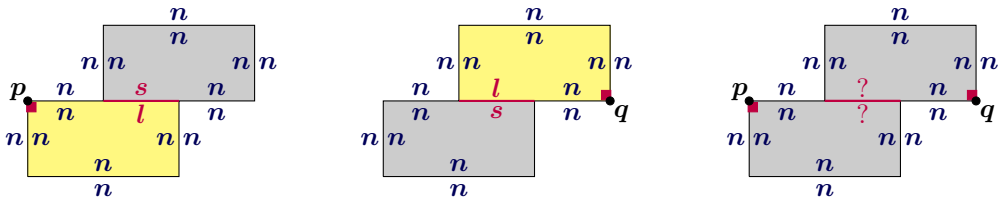


Figure 5.3: AVPs in a polygon for 2π-guards. Shadow and light AVPs have color ● and ●, respectively. The l/s and n assignments are shown as well.

There still remains the subtle case where the boundaries of two visibility polygons collide, i.e., there is some pair of edges of these polygons that collide. In these cases, if such halfedges do not have the same l/s labels, then they have to be changed to s. Actually, when this happens, it means that the faces that share these halfedges, f and f', have $\mathcal{S}(f) \not\subset \mathcal{S}(f')$ and $\mathcal{S}(f') \not\subset \mathcal{S}(f)$.

Figure 5.4 shows a simple example of those cases. If the candidate guard set for this polygon contains only p and q, then the two unique AVPs are shadows and, therefore, both guards have to used to cover the polygon.



(a) (b) (c)

Figure 5.4: An example of colliding halfedges and the analysis of the l/s labels. (a) The overlay of $\frac{\pi}{2}$-guard at vertex p. (b) The overlay of $\frac{\pi}{2}$-guard at vertex q. (c) The overlay of both $\frac{\pi}{2}$-guards at vertices p and q.

Once the shadow AVPs are identified, we find the witness points. Each is defined as the centroid of a triangle inside the corresponding shadow AVP. Then, we search for the set of guards that see each of these witnesses.

A guard g sees a witness point w if the intersection between w and the non-degenerate polygon that is visible to g is not empty. To perform this intersection, we use <CGAL/ Nef_polyhedron_2.h> library, from the *2D Boolean Operations on Nef Polygons*.

5.2.2 Finding the Conflict Graph

As we mentioned, to define $\mathcal{I}$, in the preprocessing phase, we determine the *conflict graph* $(\mathcal{G}, \mathcal{P})$. Two α-guards g_1 and g_2 are connected by an edge in this graph if the intersection between the non-regularized (i.e., degenerate) visibility polygons of g_1 and g_2 is not empty.

The set $\mathcal{I}$ consists of the sets $\{g_1, g_2\}$ of (possible) conflicting guards, defined by $\mathcal{P}$, and sets of guards that define a partition of the conflict graph into disjoint cliques. For computing that partition, we implemented a greedy algorithm described in [4]. We present it in pseudocode as Algorithm 20.

Algorithm 20 Finding a partition of a graph into disjoint cliques.

```
1:  procedure GetDisjointCliques(V_G, E_G)
2:      D ← ∅; i ← 1;
3:      while V_G ≠ ∅ do
4:          v ← GetAVertex(V_G);
5:          C_i ← {v};
6:          while ExistCommonNeighbors(V_G,E_G,C_i) do
7:              v ← GetMaximalNeighbor(V_G,E_G,C_i);
8:              C_i ← C_i ∪ {v};
9:          end while
10:         RemoveEdges(E_G, C_i); RemoveVertices(V_G, C_i);
11:         D ← D ∪ {C_i}; i ← i + 1;
12:     end while
13: end procedure
```

The variables V_G and E_G are the sets of vertices and edges of the graph, respectively. Each C_i contains the vertices of a disjoint clique and D is the set of all C_is.

GetAVertex(V_G) selects a vertex from V_G. ExistCommonNeighbors(V_G, E_G, C_i) returns *true* if there is at least one vertex $v \notin C_i$ that is adjacent to every vertex in C_i and, otherwise, returns *false*. GetMaximalNeighbor(V_G, E_G, C_i) returns the vertex $v \notin C_i$ that is adjacent to all the vertices in C_i and has the largest number of neighbors.

RemoveVertices(V_G, C_i) and RemoveEdges(E_G, C_i) update the graph, removing all the vertices that are in C_i and all the edges incident to them.

5.3 Preliminary Experimental Results

In this section, we present the results of a preliminary experimental study of α-CAGP using path orthogonal polygons. For creating the sample instances, we apply our generator for thin grid n-ogons (see Algorithm 14) and then the stretcher (see Algorithm 8) for spacing the edges randomly.

The empirical tests showed that path orthogonal polygons requiring more than two colors seem to be very rare. In addition, we realized that our solver will take too much time for polygons with more than a hundred of vertices, preventing us from getting some statistics in due time. On the other hand, from our previous experience with Steiner path polygons and path polygons, we think that the number of vertices is not a crucial factor to the optimal number of colors, for these families. Therefore, to save some time, instead of increasing the size of the instances, we decided to increase the number of polygons in each sample.

For each $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$, we generate a sample of one thousand path orthogonal polygons with a number of vertices n between 16 and 64, and run the solver to find the optimal value of α-CAGP. The value of n and $cProb$ is chosen randomly for each instance, and for the other parameters of Algorithm 8, $maxCoord$ and $sTimes$, it is assigned a random number between $\frac{n}{2}$ and $4n$ each.

The upper bound κ on the number of colors is defined as 4. We conjecture that any path is 4-colorable and that there are paths requiring this number of colors, but we could not prove this conjecture.

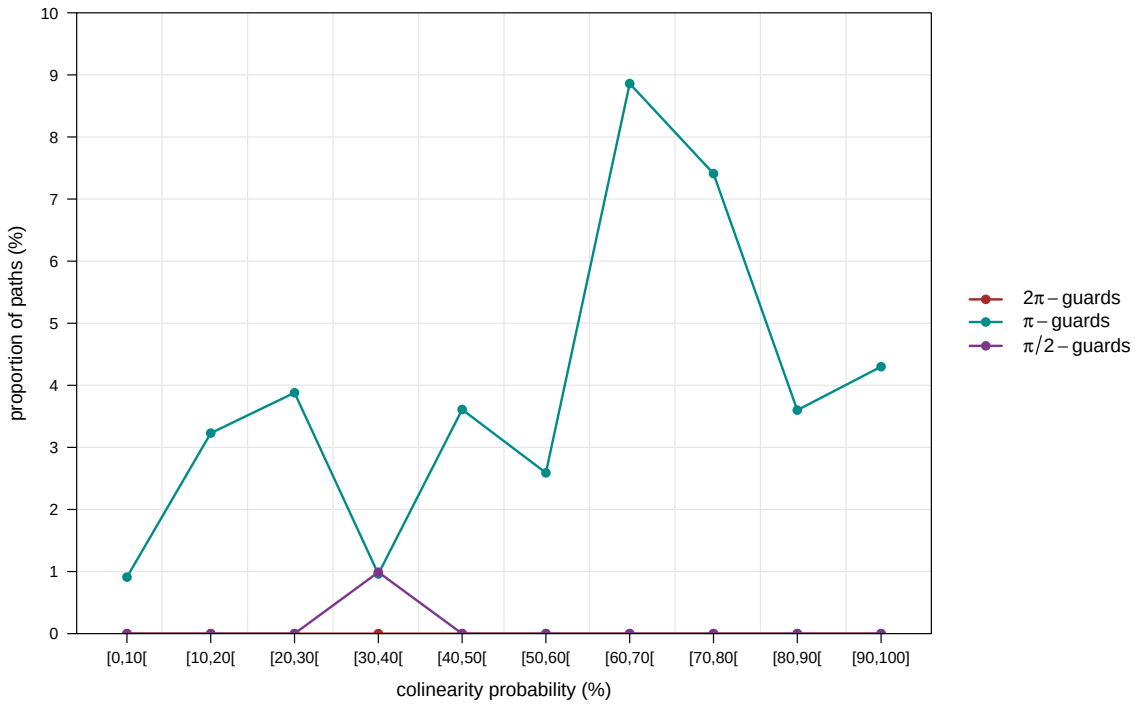
Indeed, the results were not flashy. We did not find any path requiring four colors and even paths not 2-colorable were very uncommon. We found only one path P, with 30 vertices, with $\mathcal{X}_{\mathcal{G}_{\frac{\pi}{2}}}(P) = 3$ and not even one P' with $\mathcal{X}_{\mathcal{G}_{2\pi}}(P') = 3$. For $\alpha = \pi$, the results were more interesting. We found paths with just 18 vertices not 2-colorable for edge-aligned vertex π-guards.

The results are shown graphically in Figure 5.5. The graphics describe the percentage of *paths* in a particular interval that require more than two colors. In Figure 5.5(a), we sliced each sample in ten intervals, using the probability of collinearities (i.e., the *cProb* parameter of Algorithm 8). While in Figure 5.5(b), we sliced them in six intervals, using the number of vertices.

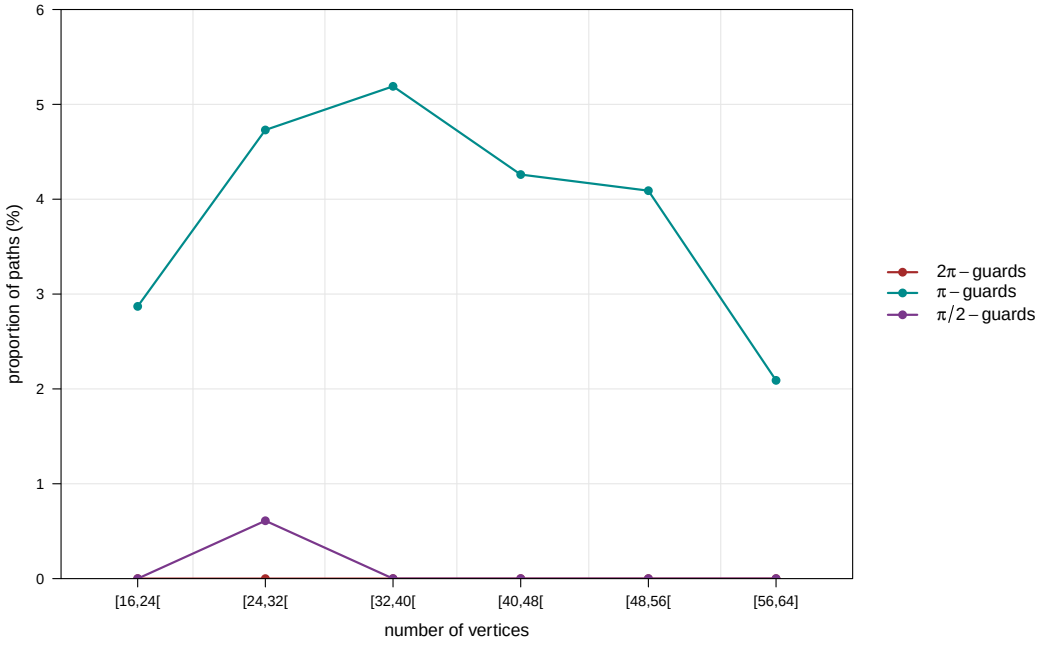
As we can see in Figure 5.5(b), the number of vertices does not seem to affect too much the number of colors and, in these samples, there is a higher percentage of paths not 2-colorable for n in the range 24 to 40. In Figure 5.5(a), the results showed some irregularities for edge-aligned vertex π-guards. This is expected somehow, because the collinearity probability acts only when a particular edge has the possibility to reach the level of higher edges. Nevertheless, we emphasize that, observing $[60, 80[$, a greater number of collinear edges may increase the number of paths requiring more than two colors.

We also performed some experiments with samples of Steiner paths, with 500 instances for each $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$. We were trying to find polygons that were not 2-colorable, with less vertices than the ones we obtained in Section 4.2. For the generation, the possible values of each parameter were the same as the previous samples of thousand of instances, unless *cProb* which had to be 0.

In the end, we could not found even one Steiner path orthogonal polygon requiring three colors in these samples. Therefore, the work we did by hand to find the instances presented in Section 4.2, was not in vain. The polygons in this class that require three colors seem really rare, and therefore are difficult to be found simply at hazard.



(a)



(b)

Figure 5.5: Proportion of paths requiring a 3-coloring. Paths organized by: (a) collinearity probability, (b) number of vertices.

Chapter 6

Conclusion

It was a great experience and gratifying to work on the field of Computational Geometry. The research work that led to the algorithms for generating the various subclasses of rectilinear polygons and to the combinatorial bounds for α-CAGP in Steiner path polygons was the most interesting part.

We developed a generator for n-ogons, as well as, for the subclasses of the row-convex, convex, path and spiral n-ogons. These series of generators are minimally well settled to be added at CGAL, a computational geometry C++ library with the latest version 4.9 beta1. We would like to try to analyze samples of polygons given by those generators to study statistically the minimum number of colors required in α-CAGP for the correspondent population of polygons, for $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$. However, unfortunately, we did not have much time to improve the prototype solver, so as to reduce the runtime. Nevertheless, we ended up with some preliminary results about path orthogonal polygons. The implementation of the discretization procedure was tricky because we also had a problem with *2D Boolean Operations on Nef Polygons* package, and it costed us some time to understand it. The problem was that the input polygons should be saved in the structure in CCW order or, otherwise, the resultant intersections among them are not well calculated. This package was very useful for us because we needed to obtain non-regularized intersections between visibility polygons to get the exact conflicts.

Despite of the problem above, we had exact results of α-CAGP, for $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$, in a particular subclass of rectilinear polygons, the Steiner paths or even thin grid n-ogons. We proved that any Steiner path P has $\mathcal{X}_{\mathcal{G}_\alpha}(P) \leq 3$ and that there exists a family of Steiner paths requiring the three colors, for each $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$. Discovering if two colors were or were not sufficient for these polygons was not easy to get for $\alpha = \frac{\pi}{2}$. We created the examples by hand, since by that time we had not developed the solver for α-CAGP yet. Otherwise, to look for instances, we could have tried to run it with Steiner path instances created by our thin grid n-ogon generator. If we had first attempted to work on the α-CAGP solver before hand, we could possibly have saved some time and eventually have the statistical results for other n-ogon subclasses.

6.1 Future Work

In the near future, we plan to improve the efficiency of our α-CAGP solver, in order to be able to carry out statistical studies on subclasses of orthogonal polygons.

In the same way, we think it would be worthwhile to look for new heuristics for creating generic orthogonal polygons from grid ogons, to cater for the unbalancing problems and collinearity probability in a better way. Currently, *cProb* parameter defines the probability that the shifted edge becomes collinear with a higher one when we move it. As the edges are moved one by one and the same edge can be moved several times, collinearities may be broken in subsequent steps.

We plan to pursue our ongoing work on combinatorial bounds for α-CAGP on path orthogonal polygons. We have the following conjecture.

Conjecture 1. *Four colors are sufficient and sometimes necessary for path orthogonal polygons, using edge-aligned vertex α-guards with $\alpha = \{\frac{\pi}{2}, \pi, 2\pi\}$.*

We have already a sketch of an inductive proof that $\mathcal{X}_{\mathcal{G}_{\frac{\pi}{2}}}(P) \leq 4$, using a similar turn-piece decomposition. We did not include it in the thesis because we still need some time to check that there is no flaw. Indeed, the number of cases we have to consider makes the proof more complex than the one we did for Steiner path orthogonal polygons. We can observe that, for example, there are nine cases of three consecutive turn-pieces for

path ogons instead of the four cases for Steiner path ogons (see Figure 6.1). We would like to find a simpler proof.

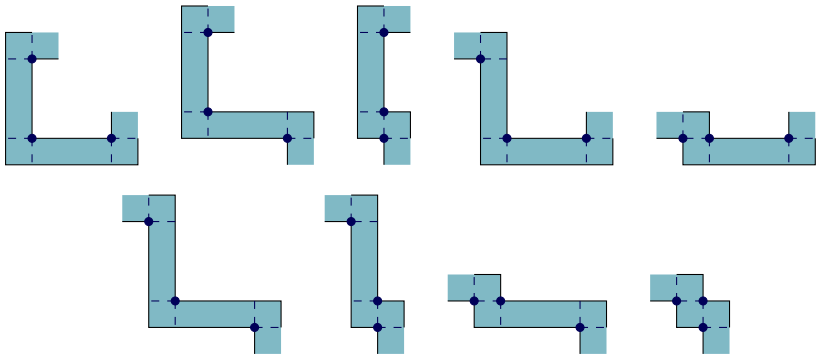


Figure 6.1: The nine cases of three consecutive turn-pieces for path ogons.

We believe that, for each $\alpha \in \{\frac{\pi}{2}, \pi, 2\pi\}$, there is a path orthogonal polygon P that $\mathcal{X}_{\mathcal{G}_\alpha}(P) \geq 4$. We think that some instances may have a section that entails a 4-coloring in a situation similar to the one we show in Figure 6.2, but we have not found such an instance.

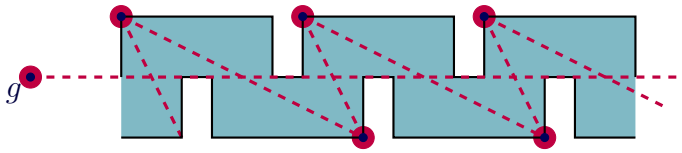


Figure 6.2: A fragment of a path ogon that may need three colors to be completely covered and a guard at g induces the fourth color.

Some of the results presented in the thesis are reported in [31].

Bibliography

[1] J. Abello, V. Estivill-Castro, T. Shermer, and J. Urrutia. "Illumination with Orthogonal Floodlights". *Algorithms and Computations: 6th International Symposium (ISAAC '95)*. Springer, 1995, pp. 362–371.

[2] A. L. Bajuelos, A. P. Tomás, and F. Marques. "Partitioning Orthogonal Polygons by Extension of All Edges Incident to Reflex Vertices: lower and upper bounds on the number of pieces". *Computational Science and Its Applications (ICCSA 2004)*. Springer, 2004, pp. 127–136.

[3] M. Bousquet-Mélou, A. J. Guttmann, W. P. Orrick, and A. Rechnitzer. "Inversion Relations, Reciprocity and Polyominoes". 3.2 (1999), pp. 223–249.

[4] S. Butenko, P. Pardalos, I. Sergienko, V. Shylo, and P. Stetsyuk. "Estimating the Size of Correcting Codes Using Extremal Graph Problems". *Optimization: Structure and Applications*. Springer, 2009, pp. 227–243.

[5] V. Chvátal. "A Combinatorial Theorem in Plane Geometry". *Journal of Combinatorial Theory Series B* 18.1 (1975), pp. 39–41.

[6] M. C. Couto, P. J. de Rezende, and C. C. de Souza. "An exact algorithm for minimizing vertex guards on art galleries". *International Transactions in Operational Research (ITOR)* 18.4 (2011), pp. 425–448.

[7] L. H Erickson. "Visibility Analysis of Landmark-Based Navigation". PhD thesis. University of Illinois at Urbana-Champaign, 2014.

[8] L. H. Erickson and S. M. LaValle. *A Chromatic Art Gallery Problem*. Tech. rep. University of Illinois at Urbana-Champaign, 2010.

[9] L. H. Erickson and S. M. LaValle. "An art gallery approach to ensuring that landmarks are distinguishable". *Proc. Robotics: Science and Systems*. 2011.

[10] L. H. Erickson and S. M. LaValle. "How many landmark colors are needed to avoid confusion in a polygon?" *Proc. 2011 IEEE International Conference on Robotics and Automation (ICRA 2011)*. 2011, pp. 2302–2307.

[11] V. Estivill-Castro and J. Urrutia. "Optimal Floodlight Illumination of Orthogonal Art Galleries". *Proc. 6th Canadian Conference on Computational Geometry (CCCG 1994)*. 1994, pp. 81–86.

[12] V. Estivill-Castro, J. O'Rourke, J. Urrutia, and D. Xu. "Illumination of Polygons with Vertex Lights". *Information Processing Letters* 56.1 (1994), pp. 9–13.

[13] S. P. Fekete, S. Friedrichs, and M. Hemmer. "Complexity of the General Chromatic Art Gallery Problem". *CoRR* abs/1403.2972 (2014).

[14] S. P. Fekete, S. Friedrichs, M. Hemmer, J. B. M. Mitchell, and C. Schmidt. "On the Chromatic Art Gallery Problem". *Proc. 26th Canadian Conference on Computational Geometry (CCCG 2014)*. 2014, pp. 73–79.

[15] S. Fisk. "A Short Proof of Chvátal's Watchman Theorem". *Journal of Combinatorial Theory Series B* 24.3 (1978), pp. 374–375.

[16] S. K. Ghosh. "Approximation Algorithms for Art Gallery Problems". *Proc. Canadian Information Processing Society Congress*. 1987, pp. 429–434.

[17] S. K. Ghosh. "Approximation Algorithms for Art Gallery Problems in Polygons". *Discrete Applied Mathematics* 158.6 (2010), pp. 718–722.

[18] A. R. M. Gonçalves. "Sobre Iluminação de Polígonos com Focos ou Reflectores em Vértices". MA thesis. University of Aveiro, 2007.

[19] R. Honsberger. *Mathematical Gems II*. Mathematical Association of America, 1976.

[20] H. Hoorfar and A. Mohades. "Special Guards in Chromatic Art Gallery". *European Workshop on Computational Geometry (EuroCG 2015)*. 2015, pp. 48–52.

[21] J. Kahn, M. Klawe, and D. Kleitman. "Traditional Art Galleries Require Fewer Watchmen". *SIAM Journal on Algebraic and Discrete Methods* 4.2 (1983), pp. 194–206.

[22] D. T. Lee and A. K. Lin. "Computational Complexity of Art Gallery Problems". *IEEE Transactions on Information Theory* 32.2 (1986), pp. 276–282.

[23] A. M. Martins. "Geometric Optimization on Visibility Problems: Metaheuristic and Exact Solutions". PhD thesis. University of Aveiro, 2009.

[24] J. O'Rourke. "An Alternate Proof of the Rectilinear Art Gallery Theorem". *Journal of Geometry* 21.1 (1983), pp. 118–130.

[25] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[26] M. H. Overmars and D. Wood. "On Rectangular Visibility". *Journal of Algorithms* 9.3 (1988), pp. 372–390.

[27] P. J. de Rezende, C. C. de Souza, S. Friedrichs, M. Hemmer, A. Kröller, and D. C. Tozoni. "Engineering Art Galleries". *CoRR* abs/1410.8720 (2014).

[28] D. Schuchardt and H.-D. Hecker. "Two NP-Hard Art-Gallery Problems for Ortho-Polygons". *Mathematical Logic Quarterly* 41.2 (1995), pp. 261–267.

[29] B. Speckmann and C. D. Tóth. "Allocating Vertex π-Guards in Simple Polygons via Pseudo-Triangulations". *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (DA03)*. 2003, pp. 109–118.

[30] A. P. Tomás and A. L. Bajuelos. "Quadratic-Time Linear-Space Algorithms for Generating Orthogonal Polygons with a Given Number of Vertices". *Computational Science and Its Applications (ICCSA 2004)*. Springer, 2004, pp. 117–126.

[31] A. P. Tomás and C. Ferreira. *On Covering Path Orthogonal Polygons*. Tech. rep. Faculty of Science of the University of Porto, 2016.

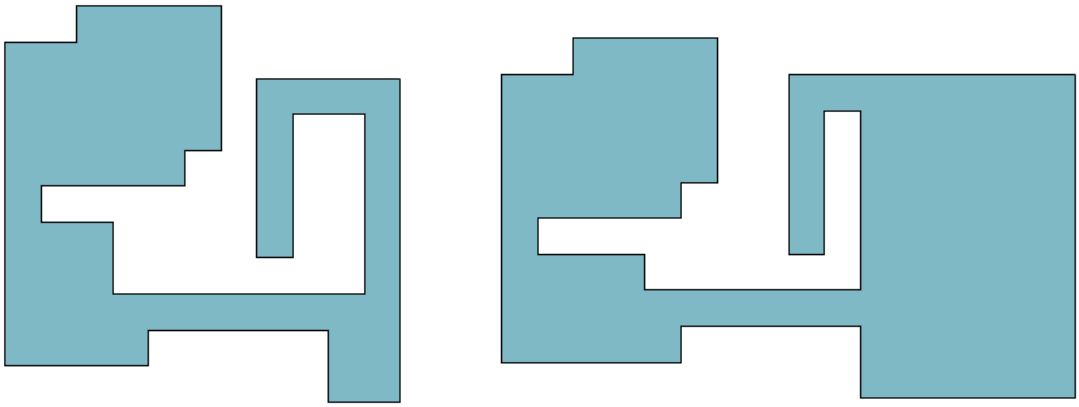
[32] J. Urrutia. "Iluminando Polígonos con Reflectores". *Proc. VI Encuentro de Geometría Computacional, Barcelona*. 1995, pp. 59–72.

[33] M. J. O. Zambon. "Soluções Exatas para o Problema Cromático da Galeria de Arte". MA thesis. University of Campinas, 2014.

[34] M. J. O. Zambon, P. J. de Rezende, and C. C. de Souza. "An Exact Algorithm for the Discrete Chromatic Art Gallery Problem". *Experimental Algorithms: 13th International Symposium (SEA 2014)*. Springer, 2014, pp. 59–73.

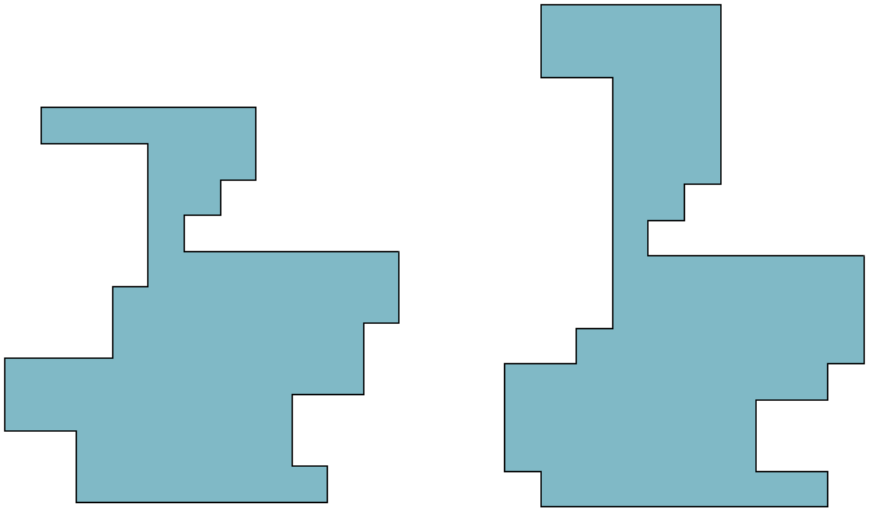
Appendix A

Some Instancies Generated by Our Implementations



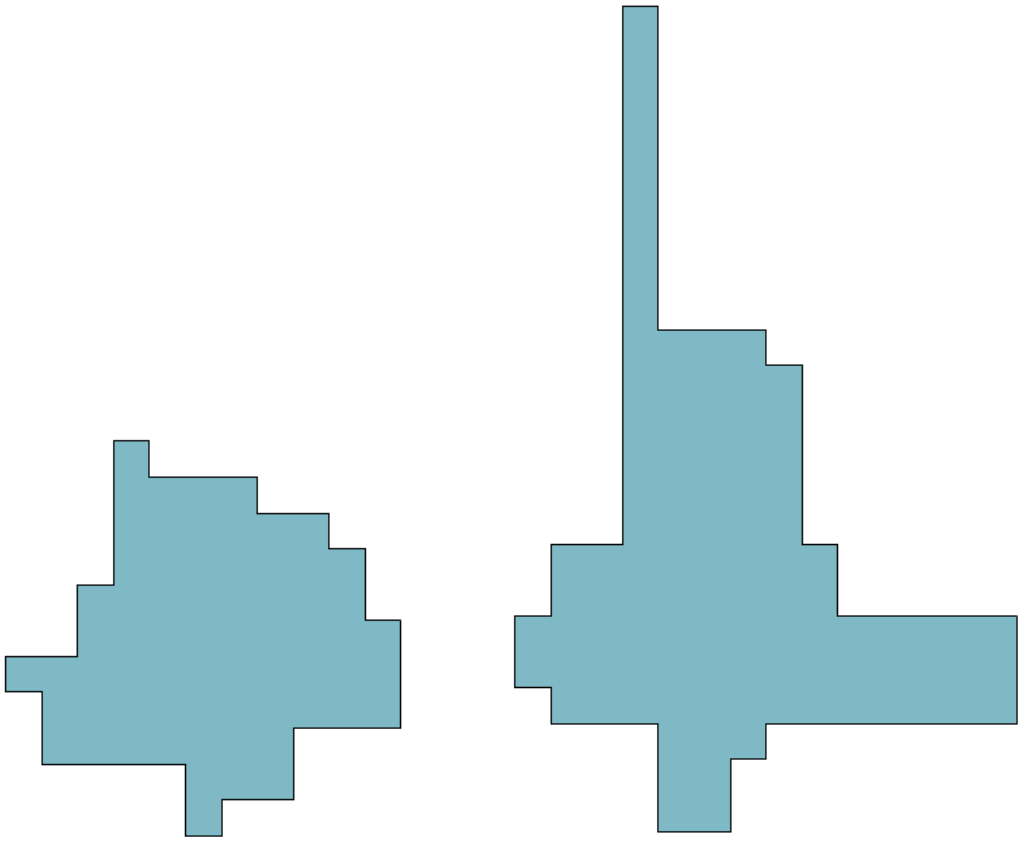
(a) (b)

Figure A.1: (a) A grid 24-ogon. (b) A 24-ogon.



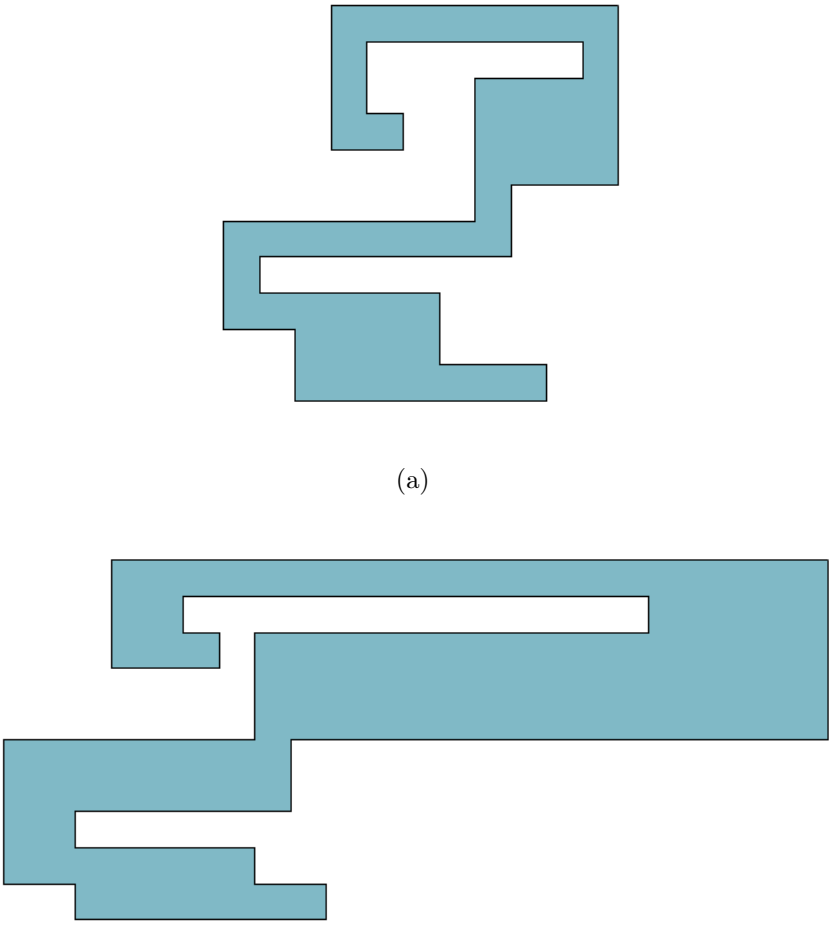
(a) (b)

Figure A.2: (a) A row-convex grid 24-ogon. (b) A row-convex 24-ogon.



(a) (b)

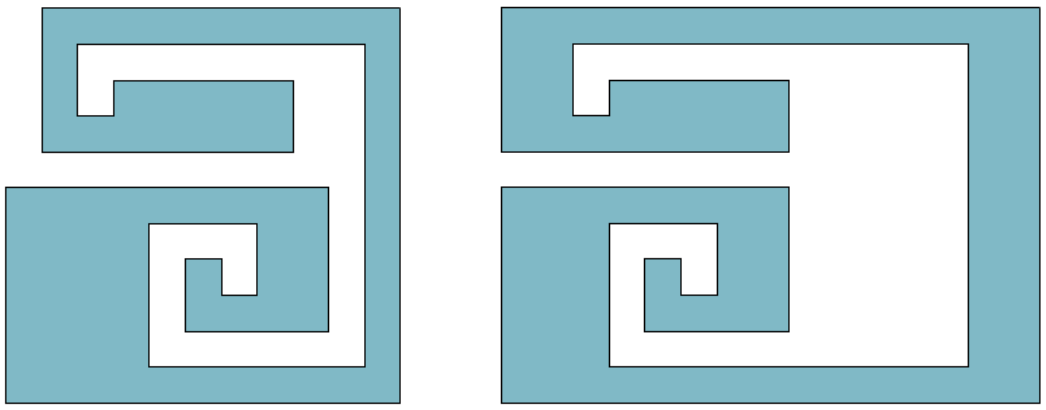
Figure A.3: (a) A convex grid 24-ogon. (b) A convex 24-ogon.



(a)

(b)

Figure A.4: (a) A thin grid 24-ogon. (b) A path 24-ogon.



(a) (b)

Figure A.5: (a) A spiral grid 24-ogon. (b) A spiral 24-ogon.

Appendix B

The Data for Space Invader

$\mathcal{C} = \{v, v' \mid v \in \{v_1, v_2, v_3, v_6, v_7, v_{10}, v_{12}, v_{13}, v_{16}\}\}$

$\mathcal{R} = \{v, v' \mid v \in \{v_4, v_5, v_8, v_9, v_{11}, v_{14}, v_{15}\}\}$

$\mathcal{K} = \{1, 2\}$

$F_1 = \{v_1, v_2, v_4^V, v_6\}, F_2 = \{v_1, v_2, v_3, v_4^H, v_4^V, v_5^H, v_6\}, F_3 = \{v_2, v_3, v_4^H, v_5^H, v_6\},$

$F_4 = \{v_3, v_4^H, v_5^H, v_6, v_8^V\}, F_5 = \{v_3, v_4^H, v_5^H, v_5^V, v_6, v_8^V, v_{10}\},$

$F_6 = \{v_3, v_5^V, v_8^V, v_9^H, v_{10}\}, F_7 = \{v_3, v_4^H, v_5^V, v_8^V, v_{10}\},$

$F_8 = \{v_3, v_5^V, v_7, v_8^H, v_8^V, v_9^H, v_{10}\}, F_9 = \{v_5^V, v_7, v_8^H, v_9^H, v_{10}\},$

$F_{10} = \{v_7, v_8^H, v_9^H, v_{10}, v_{13}\}, F_{11} = \{v_7, v_8^H, v_9^H, v_9^V, v_{10}, v_{11}^V, v_{13}\},$

$F_{12} = \{v_7, v_9^V, v_{11}^V, v_{13}\}, F_{13} = \{v_7, v_9^V, v_{11}^H, v_{11}^V, v_{12}, v_{13}, v_{14}^H\},$

$F_{14} = \{v_7, v_{11}^H, v_{12}, v_{13}, v_{14}^H\}, F_{15} = \{v_{11}^H, v_{12}, v_{13}, v_{14}^H, v_{16}\},$

$F_{16} = \{v_{11}^H, v_{12}, v_{13}, v_{14}^H, v_{14}^V, v_{15}^V, v_{16}\}, F_{17} = \{v_{12}, v_{14}^V, v_{15}^V, v_{16}\},$

$F_{18} = \{v_{12}, v_{14}^V, v_{15}^H, v_{15}^V, v_{16}, v_{15}'^H, v_{16}'\}, F_{19} = \{v_{15}^H, v_{16}, v_{15}'^H, v_{16}'\}$

$\mathcal{F} = \{F_i, F_i' \mid 1 \leq i \leq 18\} \cup \{F_{19}\}$

$I_1 = \{v_1, v_2, v_4^V, v_6\}, I_2 = \{v_1, v_2, v_4^V, v_5^H, v_6, v_8^V, v_9^H, v_{13}\},$

$I_3 = \{v_3, v_4^H, v_5^H, v_5^V, v_6, v_8^V, v_{10}\}, I_4 = \{v_1, v_2, v_3, v_4^H, v_4^V, v_5^H, v_6, v_8^V, v_9^H, v_{13}\},$

$I_5 = \{v_2, v_3, v_4^H, v_5^H, v_5^V, v_6, v_8^V, v_9^H, v_{10}, v_{13}\}, I_6 = \{v_1, v_2, v_3, v_4^H, v_4^V, v_5^H, v_6\},$

$I_7 = \{v_7, v_8^H, v_9^H, v_9^V, v_{10}, v_{11}^V, v_{13}, v_{14}^H, v_{16}\},$

$I_8 = \{v_2, v_3, v_4^H, v_5^V, v_7, v_8^H, v_8^V, v_9^H, v_{10}, v_{13}\},$

$I_9 = \{v_2, v_4^H, v_5^V, v_7, v_8^H, v_9^H, v_9^V, v_{10}, v_{11}^V, v_{13}\}, I_{10} = \{v_3, v_5^V, v_7, v_8^H, v_8^V, v_9^H, v_{10}\},$

$I_{11} = \{v_7, v_9^V, v_{11}^H, v_{11}^V, v_{12}, v_{13}, v_{14}^H, v_{16}\}, I_{12} = \{v_{11}^H, v_{12}, v_{13}, v_{14}^H, v_{14}^V, v_{15}^V, v_{16}, v_{16}'\},$

$I_{13} = \{v_2, v_4^H, v_5^V, v_7, v_8^H, v_9^V, v_{11}^H, v_{11}^V, v_{12}, v_{13}, v_{14}^H\},$

$I_{14} = \{v_7, v_{11}^H, v_{12}, v_{13}, v_{14}^H, v_{14}^V, v_{15}^V, v_{16}\}, I_{15} = \{v_{12}, v_{14}^V, v_{15}^H, v_{15}^V, v_{16}, v_{15}'^H, v_{16}'\},$

$I_{16} = \{v_7, v_{11}^H, v_{12}, v_{14}^V, v_{15}^H, v_{15}^V, v_{16}, v_{12}', v_{15}'^H, v_{16}'\}$

$\mathcal{I} = \{I_i, I_i' \mid 1 \leq i \leq 16\}$

Figure B.1: Data for "space invader", with edge-aligned vertex π-guards.

$\mathcal{C} = \{v, v' \mid v \in \{v_1, v_2, v_3, v_6, v_7, v_{10}, v_{12}, v_{13}, v_{16}\}\}$

$\mathcal{R} = \{v, v' \mid v \in \{v_4, v_5, v_8, v_9, v_{11}, v_{14}, v_{15}\}\}$

$\mathcal{K} = \{1, 2\}$

$F_1 = \{v_1, v_2, v_4^V, v_6\}, F_2 = \{v_1, v_2, v_3, v_5^H, v_6\}, F_3 = \{v_2, v_3, v_4^H, v_5^H, v_6\},$

$F_4 = \{v_3, v_4^H, v_5^H, v_6, v_8^V\}, F_5 = \{v_3, v_4^H, v_6, v_8^V, v_{10}\}, F_6 = \{v_3, v_5^V, v_8^V, v_9^H, v_{10}\},$

$F_7 = \{v_3, v_4^H, v_5^V, v_8^V, v_{10}\}, F_8 = \{v_3, v_5^V, v_7, v_9^H, v_{10}\}, F_9 = \{v_5^V, v_7, v_8^H, v_9^H, v_{10}\},$

$F_{10} = \{v_7, v_8^H, v_9^H, v_{10}, v_{13}\}, F_{11} = \{v_7, v_8^H, v_{10}, v_{11}^V, v_{13}\}, F_{12} = \{v_7, v_9^V, v_{11}^V, v_{13}\},$

$F_{13} = \{v_7, v_9^V, v_{12}, v_{13}, v_{14}^H\}, F_{14} = \{v_7, v_{11}^H, v_{12}, v_{13}, v_{14}^H\}, F_{15} = \{v_{11}^H, v_{12}, v_{13}, v_{14}^H, v_{16}\},$

$F_{16} = \{v_{11}^H, v_{12}, v_{13}, v_{15}^V, v_{16}\}, F_{17} = \{v_{12}, v_{14}^V, v_{15}^V, v_{16}\}, F_{18} = \{v_{12}, v_{14}^V, v_{16}, v_{15}'^H, v_{16}'\},$

$F_{19} = \{v_{15}^H, v_{16}, v_{15}'^H, v_{16}'\}$

$\mathcal{F} = \{F_i,\ F_i' \mid 1 \le i \le 18\} \cup \{F_{19}\}$

$I_1 = \{v_1, v_2, v_4^V, v_6\}, I_2 = \{v_1, v_2, v_4^V, v_5^H, v_6, v_8^V, v_9^H, v_{13}\}, I_3 = \{v_3, v_4^H, v_6, v_8^V, v_{10}\},$

$I_4 = \{v_1, v_2, v_3, v_4^H, v_4^V, v_5^H, v_6, v_8^V, v_9^H, v_{13}\}, I_5 = \{v_2, v_3, v_4^H, v_5^H, v_5^V, v_6, v_8^V, v_9^H, v_{10}, v_{13}\},$

$I_6 = \{v_1, v_2, v_3, v_5^H, v_6\}, I_7 = \{v_7, v_8^H, v_{10}, v_{11}^V, v_{13}, v_{14}^H, v_{16}\},$

$I_8 = \{v_2, v_3, v_4^H, v_5^V, v_7, v_8^H, v_8^V, v_9^H, v_{10}, v_{13}\}, I_9 = \{v_2, v_4^H, v_5^V, v_7, v_8^H, v_9^H, v_9^V, v_{10}, v_{11}^V, v_{13}\},$

$I_{10} = \{v_3, v_5^V, v_7, v_9^H, v_{10}\}, I_{11} = \{v_7, v_9^V, v_{11}^H, v_{11}^V, v_{12}, v_{13}, v_{14}^H, v_{16}\},$

$I_{12} = \{v_{11}^H, v_{12}, v_{13}, v_{15}^V, v_{16}, v_{16}'\}, I_{13} = \{v_2, v_4^H, v_5^V, v_7, v_8^H, v_9^V, v_{12}, v_{13}, v_{14}^H\},$

$I_{14} = \{v_7, v_{11}^H, v_{12}, v_{13}, v_{14}^H, v_{14}^V, v_{15}^V, v_{16}\}, I_{15} = \{v_{12}, v_{14}^V, v_{15}^H, v_{15}^V, v_{16}, v_{15}'^H, v_{16}'\},$

$I_{16} = \{v_7, v_{11}^H, v_{12}, v_{14}^V, v_{16}, v_{12}', v_{15}'^H, v_{16}'\}$

$\mathcal{I} = \{I_i, I_i' \mid 1 \le i \le 16\}$

Figure B.2: Data for "space invader", with edge-aligned vertex $\frac{\pi}{2}$-guards.